

# Deriving class instances for datatypes.\*

Christian Sternagel and René Thiemann

February 19, 2024

## Abstract

We provide a framework for registering automatic methods to derive class instances of datatypes, as it is possible using Haskell’s “deriving Ord, Show, . . .” feature.

We further implemented such automatic methods to derive comparators, linear orders, parametrizable equality functions, and hash-functions which are required in the Isabelle Collection Framework [1] and the Container Framework [2]. Moreover, for the tactic of Blanchette to show that a datatype is countable, we implemented a wrapper so that this tactic becomes accessible in our framework. All of the generators are based on the infrastructure that is provided by the BNF-based datatype package.

Our formalization was performed as part of the `IsaFoR/CeTA` project<sup>1</sup> [3]. With our new tactics we could remove several tedious proofs for (conditional) linear orders, and conditional equality operators within `IsaFoR` and the Container Framework.

## Contents

<b>1</b>	<b>Derive Manager</b>	<b>3</b>
<b>2</b>	<b>Shared Utilities for all Generator</b>	<b>3</b>
<b>3</b>	<b>Comparisons</b>	<b>4</b>
3.1	Comparators and Linear Orders . . . . .	4
3.2	Compare . . . . .	7
3.3	Example: Modifying the Code-Equations of Red-Black-Trees	8
3.4	A Comparator-Interface to Red-Black-Trees . . . . .	9

---

\*Supported by FWF (Austrian Science Fund) projects P27502 and Y757.

<sup>1</sup><http://cl-informatik.uibk.ac.at/software/ceta>

<b>4</b>	<b>Generating Comparators</b>	<b>18</b>
4.1	Lexicographic combination of <i>order</i> . . . . .	19
4.2	Improved code for non-lazy languages . . . . .	19
4.3	Pointwise properties for equality, symmetry, and transitivity . . . . .	19
4.4	Separate properties of comparators . . . . .	21
4.5	Auxiliary Lemmas for Comparator Generator . . . . .	23
4.6	The Comparator Generator . . . . .	23
4.7	Compare Generator . . . . .	23
4.8	Defining Comparators and Compare-Instances for Common Types . . . . .	24
4.9	Defining Compare-Order-Instances for Common Types . . . . .	26
4.10	Compare Instance for Rational Numbers . . . . .	28
4.11	Compare Instance for Real Numbers . . . . .	29
<b>5</b>	<b>Checking Equality Without "="</b>	<b>29</b>
5.1	Improved Code for Non-Lazy Languages . . . . .	30
5.2	Partial Equality Property . . . . .	30
5.3	Global equality property . . . . .	30
5.4	The Generator . . . . .	31
5.5	Defining Equality-Functions for Common Types . . . . .	31
<b>6</b>	<b>Generating Hash-Functions</b>	<b>31</b>
6.1	Improved Code for Non-Lazy Languages . . . . .	32
6.2	The Generator . . . . .	32
6.3	Defining Hash-Functions for Common Types . . . . .	32
<b>7</b>	<b>Countable Datatypes</b>	<b>32</b>
7.1	Installing the tactic . . . . .	33
<b>8</b>	<b>Loading Existing Derive-Commands</b>	<b>33</b>
<b>9</b>	<b>Examples</b>	<b>34</b>
9.1	Rational Numbers . . . . .	34
9.2	A Datatype Without Nested Recursion . . . . .	34
9.3	Using Other datatypes . . . . .	34
9.4	Mutual Recursion . . . . .	35
9.5	Nested recursion . . . . .	35
9.6	Examples from <i>IsaFoR</i> . . . . .	35
9.7	A Complex Datatype . . . . .	35
<b>10</b>	<b>Acknowledgements</b>	<b>36</b>

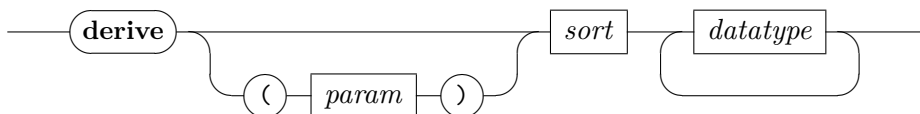
# 1 Derive Manager

```

theory Derive-Manager
imports Main
keywords print-derives :: diag and derive :: thy-decl
begin

```

The derive manager allows the user to register various derive-hooks, e.g., for orders, pretty-printers, hash-functions, etc. All registered hooks are accessible via the derive command.



**derive** (*param*) *sort datatype* calls the hook for deriving *sort* (that may depend on the optional *param*) on *datatype* (if such a hook is registered).

E.g., **derive** *compare-order list* will derive a comparator for datatype *list* which is also used to define a linear order on *lists*.

There is also the diagnostic command **print-derives** that shows the list of currently registered hooks.

```

ML-file <derive-manager.ML>

```

```

end

```

# 2 Shared Utilities for all Generator

In this theory we mainly provide some Isabelle/ML infrastructure that is used by several generators. It consists of a uniform interface to access all the theorems, terms, etc. from the BNF package, and some auxiliary functions which provide recursors on datatypes, common tactics, etc.

```

theory Generator-Aux
imports
  Main
begin

```

```

ML-file <bnf-access.ML>
ML-file <generator-aux.ML>

```

```

lemma in-set-simps:

```

$$x \in \text{set } (y \# z \# ys) = (x = y \vee x \in \text{set } (z \# ys))$$

$$x \in \text{set } ([y]) = (x = y)$$

$$x \in \text{set } [] = \text{False}$$

$$\text{Ball } (\text{set } []) P = \text{True}$$

```

Ball (set [x]) P = P x
Ball (set (x # y # zs)) P = (P x ∧ Ball (set (y # zs)) P)
by auto

```

**lemma** *conj-weak-cong*:  $a = b \implies c = d \implies (a \wedge c) = (b \wedge d)$  **by** *auto*

**lemma** *refl-True*:  $(x = x) = \text{True}$  **by** *simp*

**end**

## 3 Comparisons

### 3.1 Comparators and Linear Orders

```

theory Comparator
imports Main
begin

```

Instead of having to define a strict and a weak linear order, ( $(<)$ ,  $(\leq)$ ), one can alternative use a comparator to define the linear order, which may deliver three possible outcomes when comparing two values.

```

datatype order = Eq | Lt | Gt

```

```

type-synonym 'a comparator = 'a ⇒ 'a ⇒ order

```

In the following, we provide the obvious definitions how to switch between linear orders and comparators.

**definition** *lt-of-comp* ::  $'a \text{ comparator} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
*lt-of-comp* *acomp*  $x \ y = (\text{case } \text{acomp } x \ y \text{ of } Lt \Rightarrow \text{True} \mid - \Rightarrow \text{False})$

**definition** *le-of-comp* ::  $'a \text{ comparator} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
*le-of-comp* *acomp*  $x \ y = (\text{case } \text{acomp } x \ y \text{ of } Gt \Rightarrow \text{False} \mid - \Rightarrow \text{True})$

**definition** *comp-of-ords* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ comparator}$   
**where**  
*comp-of-ords* *le* *lt*  $x \ y = (\text{if } lt \ x \ y \text{ then } Lt \text{ else if } le \ x \ y \text{ then } Eq \text{ else } Gt)$

**lemma** *comp-of-ords-of-le-lt*[*simp*]:  $\text{comp-of-ords } (le\text{-of-comp } c) (lt\text{-of-comp } c) = c$   
**by** (*intro ext, auto simp: comp-of-ords-def le-of-comp-def lt-of-comp-def split: order.split*)

**lemma** *lt-of-comp-of-ords*:  $lt\text{-of-comp } (\text{comp-of-ords } le \ lt) = lt$   
**by** (*intro ext, auto simp: comp-of-ords-def le-of-comp-def lt-of-comp-def split: order.split*)

**lemma** *le-of-comp-of-ords-gen*:  $(\bigwedge x \ y. lt \ x \ y \implies le \ x \ y) \implies le\text{-of-comp } (\text{comp-of-ords } le \ lt) = le$   
**by** (*intro ext, auto simp: comp-of-ords-def le-of-comp-def lt-of-comp-def split: order.split*)

```

lemma le-of-comp-of-ords-linorder: assumes class.linorder le lt
  shows le-of-comp (comp-of-ords le lt) = le
proof –
  interpret linorder le lt by fact
  show ?thesis by (rule le-of-comp-of-ords-gen) simp
qed

fun invert-order:: order  $\Rightarrow$  order where
  invert-order Lt = Gt |
  invert-order Gt = Lt |
  invert-order Eq = Eq

locale comparator =
  fixes comp :: 'a comparator
  assumes sym: invert-order (comp x y) = comp y x
  and weak-eq: comp x y = Eq  $\Longrightarrow$  x = y
  and comp-trans: comp x y = Lt  $\Longrightarrow$  comp y z = Lt  $\Longrightarrow$  comp x z = Lt
begin

lemma eq: (comp x y = Eq) = (x = y)
proof
  assume x = y
  with sym [of y y] show comp x y = Eq by (cases comp x y) auto
qed (rule weak-eq)

lemma comp-same [simp]:
  comp x x = Eq
  by (simp add: eq)

abbreviation lt  $\equiv$  lt-of-comp comp
abbreviation le  $\equiv$  le-of-comp comp

sublocale ordering le lt
proof
  note [simp] = lt-of-comp-def le-of-comp-def
  fix x y z :: 'a
  show le x x using eq [of x x] by (simp)
  show le x y  $\Longrightarrow$  le y z  $\Longrightarrow$  le x z
  by (cases comp x y comp y z rule: order.exhaust [case-product order.exhaust])
  (auto dest: comp-trans simp: eq)
  show le x y  $\Longrightarrow$  le y x  $\Longrightarrow$  x = y
  using sym [of x y] by (cases comp x y) (simp-all add: eq)
  show lt x y  $\longleftrightarrow$  le x y  $\wedge$  x  $\neq$  y
  using eq [of x y] by (cases comp x y) (simp-all)
qed

lemma linorder: class.linorder le lt
proof (rule class.linorder.intro)

```

```

interpret order le lt
  using ordering-axioms by (rule ordering-orderI)
show ‹class.order le lt›
  by (fact order-axioms)
show ‹class.linorder-axioms le›
proof
  note [simp] = lt-of-comp-def le-of-comp-def
  fix x y :: 'a
  show le x y  $\vee$  le y x
    using sym [of x y] by (cases comp x y) (simp-all)
qed
qed

sublocale linorder le lt
  by (rule linorder)

lemma Gt-lt-conv: comp x y = Gt  $\longleftrightarrow$  lt y x
  unfolding lt-of-comp-def sym[of x y, symmetric]
  by (cases comp x y, auto)
lemma Lt-lt-conv: comp x y = Lt  $\longleftrightarrow$  lt x y
  unfolding lt-of-comp-def by (cases comp x y, auto)
lemma eq-Eq-conv: comp x y = Eq  $\longleftrightarrow$  x = y
  by (rule eq)
lemma nGt-le-conv: comp x y  $\neq$  Gt  $\longleftrightarrow$  le x y
  unfolding le-of-comp-def by (cases comp x y, auto)
lemma nLt-le-conv: comp x y  $\neq$  Lt  $\longleftrightarrow$  le y x
  unfolding le-of-comp-def sym[of x y, symmetric] by (cases comp x y, auto)
lemma nEq-neq-conv: comp x y  $\neq$  Eq  $\longleftrightarrow$  x  $\neq$  y
  using eq-Eq-conv[of x y] by simp

lemmas le-lt-conv = nLt-le-conv nGt-le-conv Gt-lt-conv Lt-lt-conv eq-Eq-conv
nEq-neq-conv

lemma two-comparisons-into-case-order:
  (if le x y then (if x = y then P else Q) else R) = (case-order P Q R (comp x y))
  (if le x y then (if y = x then P else Q) else R) = (case-order P Q R (comp x y))
  (if le x y then (if le y x then P else Q) else R) = (case-order P Q R (comp x y))
  (if le x y then (if lt x y then Q else P) else R) = (case-order P Q R (comp x y))
  (if lt x y then Q else (if le x y then P else R)) = (case-order P Q R (comp x y))
  (if lt x y then Q else (if lt y x then R else P)) = (case-order P Q R (comp x y))
  (if lt x y then Q else (if x = y then P else R)) = (case-order P Q R (comp x y))
  (if lt x y then Q else (if y = x then P else R)) = (case-order P Q R (comp x y))
  (if x = y then P else (if lt y x then R else Q)) = (case-order P Q R (comp x y))
  (if x = y then P else (if lt x y then Q else R)) = (case-order P Q R (comp x y))
  (if x = y then P else (if le y x then R else Q)) = (case-order P Q R (comp x y))
  (if x = y then P else (if le x y then Q else R)) = (case-order P Q R (comp x y))
  by (auto simp: le-lt-conv split: order.splits)

```

**end**

```

lemma comp-of-ords: assumes class.linorder le lt
  shows comparator (comp-of-ords le lt)
proof –
  interpret linorder le lt by fact
  show ?thesis
  by (unfold-locales, auto simp: comp-of-ords-def split: if-splits)
qed

```

```

definition (in linorder) comparator-of :: 'a comparator where
  comparator-of x y = (if x < y then Lt else if x = y then Eq else Gt)

```

```

lemma comparator-of: comparator comparator-of
  by (unfold-locales (auto split: if-splits simp: comparator-of-def))

```

**end**

## 3.2 Compare

```

theory Compare
imports Comparator
keywords compare-code :: thy-decl
begin

```

This introduces a type class for having a proper comparator, similar to *linorder*. Since most of the Isabelle/HOL algorithms work on the latter, we also provide a method which turns linear-order based algorithms into comparator-based algorithms, where two consecutive invocations of linear orders and equality are merged into one comparator invocation. We further define a class which both define a linear order and a comparator, and where the induces orders coincide.

```

class compare =
  fixes compare :: 'a comparator
  assumes comparator-compare: comparator compare
begin

```

```

lemma compare-Eq-is-eq [simp]:
  compare x y = Eq  $\longleftrightarrow$  x = y
  by (rule comparator.eq [OF comparator-compare])

```

```

lemma compare-refl [simp]:
  compare x x = Eq
  by simp

```

**end**

```

lemma (in linorder) le-lt-comparator-of:
  le-of-comp comparator-of = ( $\leq$ ) lt-of-comp comparator-of = (<)
  by (intro ext, auto simp: comparator-of-def le-of-comp-def lt-of-comp-def)+

```

```

class compare-order = ord + compare +
  assumes ord-defs: le-of-comp compare = ( $\leq$ ) lt-of-comp compare = ( $<$ )

  compare-order is compare and linorder, where comparator and orders
  define the same ordering.

subclass (in compare-order) linorder
  by (unfold ord-defs[symmetric], rule comparator.linorder, rule comparator-compare)

context compare-order
begin

lemma compare-is-comparator-of:
  compare = comparator-of
proof (intro ext)
  fix x y :: 'a
  show compare x y = comparator-of x y
  by (unfold comparator-of-def, unfold ord-defs[symmetric] lt-of-comp-def,
    cases compare x y, auto)
qed

lemmas two-comparisons-into-compare =
  comparator.two-comparisons-into-case-order[OF comparator-compare, unfolded ord-defs]

thm two-comparisons-into-compare
end

ML-file <compare-code.ML>

  Compare-Code.change-compare-code const ty-vars changes the code equations
  of some constant such that two consecutive comparisons via ( $\leq$ ), ( $<$ ),
  or ( $=$ ) are turned into one invocation of compare. The difference to a standard
  code-unfold is that here we change the code-equations where an additional
  sort-constraint on compare-order can be added. Otherwise, there would be
  no compare-function.

end

```

### 3.3 Example: Modifying the Code-Equations of Red-Black-Trees

```

theory RBT-Compare-Order-Impl
imports
  Compare
  HOL-Library.RBT-Impl
begin

```

In the following, we modify all code-equations of the red-black-tree implementation that perform comparisons. As a positive result, they now all require one invocation of *comparator*, where before two comparisons have



been performed. The disadvantage of this simple solution is the additional class constraint on *compare-order*.

```

compare-code ('a) rbt-ins
compare-code ('a) rbt-lookup
compare-code ('a) rbt-del
compare-code ('a) rbt-map-entry
compare-code ('a) sunion-with
compare-code ('a) sinter-with
compare-code ('a) rbt-split

```

```

export-code rbt-ins rbt-lookup rbt-del rbt-map-entry rbt-union-with-key rbt-inter-with-key
rbt-minus in Haskell

```

**end**

### 3.4 A Comparator-Interface to Red-Black-Trees

```

theory RBT-Comparator-Impl
imports
  HOL-Library.RBT-Impl
  Comparator
begin

```

For all of the main algorithms of red-black trees, we provide alternatives which are completely based on comparators, and which are provable equivalent. At the time of writing, this interface is used in the Container AFP-entry.

It does not rely on the modifications of code-equations as in the previous subsection.

```

context
  fixes c :: 'a comparator
begin

```

```

primrec rbt-comp-lookup :: ('a, 'b) rbt  $\Rightarrow$  'a  $\rightarrow$  'b
where

```

```

  rbt-comp-lookup RBT-Impl.Empty k = None
| rbt-comp-lookup (Branch - l x y r) k =
  (case c k x of Lt  $\Rightarrow$  rbt-comp-lookup l k
   | Gt  $\Rightarrow$  rbt-comp-lookup r k
   | Eq  $\Rightarrow$  Some y)

```

```

fun
  rbt-comp-ins :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a,'b) rbt  $\Rightarrow$  ('a,'b) rbt
where
  rbt-comp-ins f k v RBT-Impl.Empty = Branch RBT-Impl.R RBT-Impl.Empty k
v RBT-Impl.Empty |
  rbt-comp-ins f k v (Branch RBT-Impl.B l x y r) = (case c k x of
    Lt  $\Rightarrow$  balance (rbt-comp-ins f k v l) x y r

```

$| Gt \Rightarrow \text{balance } l \ x \ y \ (\text{rbt-comp-ins } f \ k \ v \ r)$   
 $| Eq \Rightarrow \text{Branch } RBT\text{-Impl.}B \ l \ x \ (f \ k \ y \ v) \ r \ |$   
 $\text{rbt-comp-ins } f \ k \ v \ (\text{Branch } RBT\text{-Impl.}R \ l \ x \ y \ r) = (\text{case } c \ k \ x \ \text{of}$   
 $\quad Lt \Rightarrow \text{Branch } RBT\text{-Impl.}R \ (\text{rbt-comp-ins } f \ k \ v \ l) \ x \ y \ r$   
 $\quad | Gt \Rightarrow \text{Branch } RBT\text{-Impl.}R \ l \ x \ y \ (\text{rbt-comp-ins } f \ k \ v \ r)$   
 $\quad | Eq \Rightarrow \text{Branch } RBT\text{-Impl.}R \ l \ x \ (f \ k \ y \ v) \ r)$

**definition**  $\text{rbt-comp-insert-with-key} :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

**where**  $\text{rbt-comp-insert-with-key } f \ k \ v \ t = \text{paint } RBT\text{-Impl.}B \ (\text{rbt-comp-ins } f \ k \ v \ t)$

**definition**  $\text{rbt-comp-insert} :: 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$  **where**

$\text{rbt-comp-insert} = \text{rbt-comp-insert-with-key } (\lambda - \text{. } nv. \ nv)$

**fun**

$\text{rbt-comp-del-from-left} :: 'a \Rightarrow ('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

**and**

$\text{rbt-comp-del-from-right} :: 'a \Rightarrow ('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

**and**

$\text{rbt-comp-del} :: 'a \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

**where**

$\text{rbt-comp-del } x \ RBT\text{-Impl.}Empty = RBT\text{-Impl.}Empty \ |$

$\text{rbt-comp-del } x \ (\text{Branch } - \ a \ y \ s \ b) =$

$(\text{case } c \ x \ y \ \text{of}$

$\quad Lt \Rightarrow \text{rbt-comp-del-from-left } x \ a \ y \ s \ b$

$\quad | Gt \Rightarrow \text{rbt-comp-del-from-right } x \ a \ y \ s \ b$

$\quad | Eq \Rightarrow \text{combine } a \ b) \ |$

$\text{rbt-comp-del-from-left } x \ (\text{Branch } RBT\text{-Impl.}B \ lt \ z \ v \ rt) \ y \ s \ b = \text{balance-left}$   
 $(\text{rbt-comp-del } x \ (\text{Branch } RBT\text{-Impl.}B \ lt \ z \ v \ rt)) \ y \ s \ b \ |$

$\text{rbt-comp-del-from-left } x \ a \ y \ s \ b = \text{Branch } RBT\text{-Impl.}R \ (\text{rbt-comp-del } x \ a) \ y \ s \ b \ |$

$\text{rbt-comp-del-from-right } x \ a \ y \ s \ (\text{Branch } RBT\text{-Impl.}B \ lt \ z \ v \ rt) = \text{balance-right } a$   
 $y \ s \ (\text{rbt-comp-del } x \ (\text{Branch } RBT\text{-Impl.}B \ lt \ z \ v \ rt)) \ |$

$\text{rbt-comp-del-from-right } x \ a \ y \ s \ b = \text{Branch } RBT\text{-Impl.}R \ a \ y \ s \ (\text{rbt-comp-del } x \ b)$

**definition**  $\text{rbt-comp-delete } k \ t = \text{paint } RBT\text{-Impl.}B \ (\text{rbt-comp-del } k \ t)$

**definition**  $\text{rbt-comp-bulkload } xs = \text{foldr } (\lambda(k, v). \ \text{rbt-comp-insert } k \ v) \ xs \ RBT\text{-Impl.}Empty$

**primrec**

$\text{rbt-comp-map-entry} :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

**where**

$\text{rbt-comp-map-entry } k \ f \ RBT\text{-Impl.}Empty = RBT\text{-Impl.}Empty$

$| \text{rbt-comp-map-entry } k \ f \ (\text{Branch } cc \ lt \ x \ v \ rt) =$

$(\text{case } c \ k \ x \ \text{of}$

$\quad Lt \Rightarrow \text{Branch } cc \ (\text{rbt-comp-map-entry } k \ f \ lt) \ x \ v \ rt$

$\quad | Gt \Rightarrow \text{Branch } cc \ lt \ x \ v \ (\text{rbt-comp-map-entry } k \ f \ rt)$

$\quad | Eq \Rightarrow \text{Branch } cc \ lt \ x \ (f \ v) \ rt)$

**function**  $\text{comp-union-with} :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \times 'b) \text{list} \Rightarrow ('a \times 'b)$

$list \Rightarrow ('a \times 'b) list$

**where**

$comp\text{-}sunion\text{-}with\ f\ ((k, v) \# as)\ ((k', v') \# bs) =$   
 $(case\ c\ k'\ k\ of$   
   $Lt \Rightarrow (k', v') \# comp\text{-}sunion\text{-}with\ f\ ((k, v) \# as)\ bs$   
   $| Gt \Rightarrow (k, v) \# comp\text{-}sunion\text{-}with\ f\ as\ ((k', v') \# bs)$   
   $| Eq \Rightarrow (k, f\ k\ v\ v') \# comp\text{-}sunion\text{-}with\ f\ as\ bs)$

$| comp\text{-}sunion\text{-}with\ f\ []\ bs = bs$

$| comp\text{-}sunion\text{-}with\ f\ as\ [] = as$

**by** *pat-completeness auto*

**termination by** *lexicographic-order*

**function**  $comp\text{-}sinter\text{-}with :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \times 'b) list \Rightarrow ('a \times 'b)$

$list \Rightarrow ('a \times 'b) list$

**where**

$comp\text{-}sinter\text{-}with\ f\ ((k, v) \# as)\ ((k', v') \# bs) =$   
 $(case\ c\ k'\ k\ of$   
   $Lt \Rightarrow comp\text{-}sinter\text{-}with\ f\ ((k, v) \# as)\ bs$   
   $| Gt \Rightarrow comp\text{-}sinter\text{-}with\ f\ as\ ((k', v') \# bs)$   
   $| Eq \Rightarrow (k, f\ k\ v\ v') \# comp\text{-}sinter\text{-}with\ f\ as\ bs)$

$| comp\text{-}sinter\text{-}with\ f\ []\ - = []$

$| comp\text{-}sinter\text{-}with\ f\ -\ [] = []$

**by** *pat-completeness auto*

**termination by** *lexicographic-order*

**fun**  $rbt\text{-}split\text{-}comp :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow ('a, 'b) rbt \times 'b\ option \times ('a, 'b) rbt$

**where**

$rbt\text{-}split\text{-}comp\ RBT\text{-}Impl.\ Empty\ k = (RBT\text{-}Impl.\ Empty, None, RBT\text{-}Impl.\ Empty)$

$| rbt\text{-}split\text{-}comp\ (RBT\text{-}Impl.\ Branch\ -\ l\ a\ b\ r)\ x = (case\ c\ x\ a\ of$

$Lt \Rightarrow (case\ rbt\text{-}split\text{-}comp\ l\ x\ of\ (l1, \beta, l2) \Rightarrow (l1, \beta, rbt\text{-}join\ l2\ a\ b\ r))$

$| Gt \Rightarrow (case\ rbt\text{-}split\text{-}comp\ r\ x\ of\ (r1, \beta, r2) \Rightarrow (rbt\text{-}join\ l\ a\ b\ r1, \beta, r2))$

$| Eq \Rightarrow (l, Some\ b, r))$

**lemma**  $rbt\text{-}split\text{-}comp\text{-}size: (l2, b, r2) = rbt\text{-}split\text{-}comp\ t2\ a \Longrightarrow size\ l2 + size\ r2$   
 $\leq size\ t2$

**by** (*induction t2 a arbitrary: l2 b r2 rule: rbt-split-comp.induct*)

(*auto split: order.splits if-splits prod.splits*)

**function**  $rbt\text{-}comp\text{-}union\text{-}rec :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt$

$\Rightarrow ('a, 'b) rbt$  **where**

$rbt\text{-}comp\text{-}union\text{-}rec\ f\ t1\ t2 = (let\ (f, t2, t1) =$

$if\ flip\text{-}rbt\ t2\ t1\ then\ (\lambda k\ v\ v'.\ f\ k\ v'\ v, t1, t2)\ else\ (f, t2, t1)\ in$

$if\ small\text{-}rbt\ t2\ then\ RBT\text{-}Impl.\ fold\ (rbt\text{-}comp\text{-}insert\text{-}with\text{-}key\ f)\ t2\ t1$

$else\ (case\ t1\ of\ RBT\text{-}Impl.\ Empty \Rightarrow t2$

$| RBT\text{-}Impl.\ Branch\ -\ l1\ a\ b\ r1 \Rightarrow$

$case\ rbt\text{-}split\text{-}comp\ t2\ a\ of\ (l2, \beta, r2) \Rightarrow$

$rbt\text{-}join\ (rbt\text{-}comp\text{-}union\text{-}rec\ f\ l1\ l2)\ a\ (case\ \beta\ of\ None \Rightarrow b\ | Some\ b' \Rightarrow f$

$a\ b\ b')\ (rbt\text{-}comp\text{-}union\text{-}rec\ f\ r1\ r2)))$

**by** *pat-completeness auto*

**termination**  
**using** *rbt-split-comp-size*  
**by** (*relation measure* ( $\lambda(f,t1,t2). \text{size } t1 + \text{size } t2$ )) (*fastforce split: if-splits*)+

**declare** *rbt-comp-union-rec.simps*[*simp del*]

**function** *rbt-comp-union-swap-rec* :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  bool  $\Rightarrow$  ('a, 'b) *rbt*  
 $\Rightarrow$  ('a, 'b) *rbt*  $\Rightarrow$  ('a, 'b) *rbt* **where**  
*rbt-comp-union-swap-rec* *f*  $\gamma$  *t1* *t2* = (*let* ( $\gamma$ , *t2*, *t1*) =  
*if flip-rbt t2 t1 then* ( $\neg\gamma$ , *t1*, *t2*) *else* ( $\gamma$ , *t2*, *t1*);  
*f'* = (*if*  $\gamma$  *then* ( $\lambda k v v'. f k v' v$ ) *else* *f*) *in*  
*if small-rbt t2 then* *RBT-Impl.fold* (*rbt-comp-insert-with-key f'*) *t2 t1*  
*else case t1 of* *rbt.Empty*  $\Rightarrow$  *t2*  
| *Branch x l1 a b r1*  $\Rightarrow$   
*case rbt-split-comp t2 a of* (*l2*,  $\beta$ , *r2*)  $\Rightarrow$   
*rbt-join* (*rbt-comp-union-swap-rec f*  $\gamma$  *l1 l2*) *a* (*case*  $\beta$  *of* *None*  $\Rightarrow$  *b* | *Some*  
*x*  $\Rightarrow$  *f' a b x*) (*rbt-comp-union-swap-rec f*  $\gamma$  *r1 r2*))  
**by** *pat-completeness auto*

**termination**  
**using** *rbt-split-comp-size*  
**by** (*relation measure* ( $\lambda(f,\gamma,t1, t2). \text{size } t1 + \text{size } t2$ )) (*fastforce split: if-splits*)+

**declare** *rbt-comp-union-swap-rec.simps*[*simp del*]

**lemma** *rbt-comp-union-swap-rec*: *rbt-comp-union-swap-rec f*  $\gamma$  *t1* *t2* =  
*rbt-comp-union-rec* (*if*  $\gamma$  *then* ( $\lambda k v v'. f k v' v$ ) *else* *f*) *t1* *t2*  
**proof** (*induction f*  $\gamma$  *t1* *t2* *rule: rbt-comp-union-swap-rec.induct*)  
**case** (*1 f*  $\gamma$  *t1* *t2*)  
**show** ?*case*  
**using** *1*[*OF refl - refl refl - refl - refl*]  
**unfolding** *rbt-comp-union-swap-rec.simps*[*of - - t1*] *rbt-comp-union-rec.simps*[*of*  
- *t1*]  
**by** (*auto simp: Let-def split: rbt.splits prod.splits option.splits*)  
**qed**

**lemma** *rbt-comp-union-swap-rec-code*[*code*]: *rbt-comp-union-swap-rec f*  $\gamma$  *t1* *t2* =  
(  
*let* *bh1* = *bheight t1*; *bh2* = *bheight t2*; ( $\gamma$ , *t2*, *bh2*, *t1*, *bh1*) =  
*if* *bh1* < *bh2* *then* ( $\neg\gamma$ , *t1*, *bh1*, *t2*, *bh2*) *else* ( $\gamma$ , *t2*, *bh2*, *t1*, *bh1*);  
*f'* = (*if*  $\gamma$  *then* ( $\lambda k v v'. f k v' v$ ) *else* *f*) *in*  
*if* *bh2* < 4 *then* *RBT-Impl.fold* (*rbt-comp-insert-with-key f'*) *t2 t1*  
*else case t1 of* *rbt.Empty*  $\Rightarrow$  *t2*  
| *Branch x l1 a b r1*  $\Rightarrow$   
*case rbt-split-comp t2 a of* (*l2*,  $\beta$ , *r2*)  $\Rightarrow$   
*rbt-join* (*rbt-comp-union-swap-rec f*  $\gamma$  *l1 l2*) *a* (*case*  $\beta$  *of* *None*  $\Rightarrow$  *b* | *Some*  
*x*  $\Rightarrow$  *f' a b x*) (*rbt-comp-union-swap-rec f*  $\gamma$  *r1 r2*))  
**by** (*auto simp: rbt-comp-union-swap-rec.simps flip-rbt-def small-rbt-def*)

**definition** *rbt-comp-union-with-key* *f* *t1* *t2* = *paint RBT-Impl.B* (*rbt-comp-union-swap-rec*

$f$  *False*  $t1$   $t2$ )

**definition** *map-filter-comp-inter*  $f$   $t1$   $t2 = List.map-filter (\lambda(k, v).$   
*case* *rbt-comp-lookup*  $t1$   $k$  *of* *None*  $\Rightarrow$  *None*  
 $|$  *Some*  $v' \Rightarrow$  *Some*  $(k, f\ k\ v'\ v)$ ) (*RBT-Impl.entries*  $t2$ )

**function** *rbt-comp-inter-rec*  $:: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt*  
 $\Rightarrow ('a, 'b)$  *rbt* **where**  
*rbt-comp-inter-rec*  $f$   $t1$   $t2 = (let (f, t2, t1) =$   
*if* *flip-rbt*  $t2$   $t1$  *then*  $(\lambda k\ v\ v'. f\ k\ v'\ v, t1, t2)$  *else*  $(f, t2, t1)$  *in*  
*if* *small-rbt*  $t2$  *then* *rmtreeify* (*map-filter-comp-inter*  $f$   $t1$   $t2$ )  
*else* *case*  $t1$  *of* *RBT-Impl.Empty*  $\Rightarrow$  *RBT-Impl.Empty*  
 $|$  *RBT-Impl.Branch*  $-$   $l1$   $a$   $b$   $r1 \Rightarrow$   
*case* *rbt-split-comp*  $t2$   $a$  *of*  $(l2, \beta, r2) \Rightarrow$  *let*  $l' =$  *rbt-comp-inter-rec*  $f$   $l1$   $l2$ ;  $r' =$   
*rbt-comp-inter-rec*  $f$   $r1$   $r2$  *in*  
 $(case\ \beta$  *of* *None*  $\Rightarrow$  *rbt-join2*  $l'$   $r'$   $|$  *Some*  $b' \Rightarrow$  *rbt-join*  $l'$   $a$   $(f\ a\ b\ b')$   $r')$ )  
**by** *pat-completeness* *auto*  
**termination**  
**using** *rbt-split-comp-size*  
**by** (*relation measure*  $(\lambda(f,t1,t2). size\ t1 + size\ t2)$ ) (*fastforce split: if-splits*)+

**declare** *rbt-comp-inter-rec.simps*[*simp del*]

**function** *rbt-comp-inter-swap-rec*  $:: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow bool \Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow$   
 $(('a, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt* **where**  
*rbt-comp-inter-swap-rec*  $f$   $\gamma$   $t1$   $t2 = (let (\gamma, t2, t1) =$   
*if* *flip-rbt*  $t2$   $t1$  *then*  $(\neg\gamma, t1, t2)$  *else*  $(\gamma, t2, t1)$ ;  
 $f' =$  *if*  $\gamma$  *then*  $(\lambda k\ v\ v'. f\ k\ v'\ v)$  *else*  $f$  *in*  
*if* *small-rbt*  $t2$  *then* *rmtreeify* (*map-filter-comp-inter*  $f'$   $t1$   $t2$ )  
*else* *case*  $t1$  *of* *rbt.Empty*  $\Rightarrow$  *rbt.Empty*  
 $|$  *Branch*  $x$   $l1$   $a$   $b$   $r1 \Rightarrow$   
 $(case\ rbt-split-comp\ t2\ a$  *of*  $(l2, \beta, r2) \Rightarrow$  *let*  $l' =$  *rbt-comp-inter-swap-rec*  $f$   $\gamma$   
 $l1$   $l2$ ;  $r' =$  *rbt-comp-inter-swap-rec*  $f$   $\gamma$   $r1$   $r2$  *in*  
 $(case\ \beta$  *of* *None*  $\Rightarrow$  *rbt-join2*  $l'$   $r'$   $|$  *Some*  $b' \Rightarrow$  *rbt-join*  $l'$   $a$   $(f'\ a\ b\ b')$   $r'))$ )  
**by** *pat-completeness* *auto*  
**termination**  
**using** *rbt-split-comp-size*  
**by** (*relation measure*  $(\lambda(f,\gamma,t1,t2). size\ t1 + size\ t2)$ ) (*fastforce split: if-splits*)+

**declare** *rbt-comp-inter-swap-rec.simps*[*simp del*]

**lemma** *rbt-comp-inter-swap-rec*: *rbt-comp-inter-swap-rec*  $f$   $\gamma$   $t1$   $t2 =$   
*rbt-comp-inter-rec* (*if*  $\gamma$  *then*  $(\lambda k\ v\ v'. f\ k\ v'\ v)$  *else*  $f$ )  $t1$   $t2$

**proof** (*induction*  $f$   $\gamma$   $t1$   $t2$  *rule*: *rbt-comp-inter-swap-rec.induct*)

**case**  $(1\ f\ \gamma\ t1\ t2)$

**show** *?case*

**using**  $1[OF\ refl - refl\ refl - refl - refl]$

**unfolding** *rbt-comp-inter-swap-rec.simps*[*of - - t1*] *rbt-comp-inter-rec.simps*[*of - t1*]

by (auto simp: Let-def split: rbt.splits prod.splits option.splits)  
**qed**

**lemma** *comp-inter-with-key-code*[code]:  $\text{rbt-comp-inter-swap-rec } f \ \gamma \ t1 \ t2 = ($   
*let*  $\text{bh1} = \text{bheight } t1; \text{bh2} = \text{bheight } t2; (\gamma, t2, \text{bh2}, t1, \text{bh1}) =$   
*if*  $\text{bh1} < \text{bh2}$  *then*  $(\neg\gamma, t1, \text{bh1}, t2, \text{bh2})$  *else*  $(\gamma, t2, \text{bh2}, t1, \text{bh1});$   
 $f' = (\text{if } \gamma \text{ then } (\lambda k \ v \ v'. f \ k \ v' \ v) \text{ else } f)$  *in*  
*if*  $\text{bh2} < 4$  *then*  $\text{rbtreeify } (\text{map-filter-comp-inter } f' \ t1 \ t2)$   
*else case*  $t1$  *of*  $\text{rbt.Empty} \Rightarrow \text{rbt.Empty}$   
|  $\text{Branch } x \ l1 \ a \ b \ r1 \Rightarrow$   
*(case*  $\text{rbt-split-comp } t2 \ a$  *of*  $(l2, \beta, r2) \Rightarrow \text{let } l' = \text{rbt-comp-inter-swap-rec } f \ \gamma$   
 $l1 \ l2; r' = \text{rbt-comp-inter-swap-rec } f \ \gamma \ r1 \ r2$  *in*  
*(case*  $\beta$  *of*  $\text{None} \Rightarrow \text{rbt-join2 } l' \ r' \mid \text{Some } b' \Rightarrow \text{rbt-join } l' \ a \ (f' \ a \ b \ b') \ r')$   
**by** (auto simp: rbt-comp-inter-swap-rec.simps flip-rbt-def small-rbt-def)

**definition** *rbt-comp-inter-with-key*  $f \ t1 \ t2 = \text{paint } \text{RBT-Impl.B } (\text{rbt-comp-inter-swap-rec } f \ \text{False } t1 \ t2)$

**definition** *filter-comp-minus*  $t1 \ t2 =$   
*filter*  $(\lambda(k, -). \text{rbt-comp-lookup } t2 \ k = \text{None}) (\text{RBT-Impl.entries } t1)$

**fun** *comp-minus* ::  $('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$  **where**  
*comp-minus*  $t1 \ t2 = (\text{if } \text{small-rbt } t2 \text{ then } \text{RBT-Impl.fold } (\lambda k \ t. \text{rbt-comp-delete } k \ t) \ t2 \ t1$   
*else if*  $\text{small-rbt } t1$  *then*  $\text{rbtreeify } (\text{filter-comp-minus } t1 \ t2)$   
*else case*  $t2$  *of*  $\text{RBT-Impl.Empty} \Rightarrow t1$   
|  $\text{RBT-Impl.Branch } l2 \ a \ b \ r2 \Rightarrow$   
*case*  $\text{rbt-split-comp } t1 \ a$  *of*  $(l1, -, r1) \Rightarrow \text{rbt-join2 } (\text{comp-minus } l1 \ l2)$   
 $(\text{comp-minus } r1 \ r2))$

**declare** *comp-minus.simps*[simp del]

**definition** *rbt-comp-minus*  $t1 \ t2 = \text{paint } \text{RBT-Impl.B } (\text{comp-minus } t1 \ t2)$

**context**

**assumes**  $c$ : *comparator*  $c$   
**begin**

**lemma** *rbt-comp-lookup*:  $\text{rbt-comp-lookup} = \text{ord.rbt-lookup } (\text{lt-of-comp } c)$

**proof** (*intro ext*)

**fix**  $k$  **and**  $t :: ('a, 'b) \text{rbt}$

**show**  $\text{rbt-comp-lookup } t \ k = \text{ord.rbt-lookup } (\text{lt-of-comp } c) \ t \ k$

**by** (*induct*  $t$ , *unfold* *rbt-comp-lookup.simps* *ord.rbt-lookup.simps*  
*comparator.two-comparisons-into-case-order*[OF  $c$ ])  
*(auto split: order.splits)*

**qed**

**lemma** *rbt-comp-ins*:  $\text{rbt-comp-ins} = \text{ord.rbt-ins } (\text{lt-of-comp } c)$

**proof** (*intro ext*)

**fix**  $f k v$  **and**  $t :: ('a, 'b)rbt$   
**show**  $rbt\text{-comp-ins } f k v t = ord.rbt\text{-ins } (lt\text{-of-comp } c) f k v t$   
**by** (*induct*  $f k v t$  *rule*:  $rbt\text{-comp-ins.induct}$ , *unfold*  $rbt\text{-comp-ins.simps } ord.rbt\text{-ins.simps$   
*comparator.two-comparisons-into-case-order*[ $OF c$ ])  
*(auto split: order.splits)*  
**qed**

**lemma**  $rbt\text{-comp-insert-with-key}$ :  $rbt\text{-comp-insert-with-key} = ord.rbt\text{-insert-with-key}$   
*(lt-of-comp c)*  
**unfolding**  $rbt\text{-comp-insert-with-key-def}$ [*abs-def*]  $ord.rbt\text{-insert-with-key-def}$ [*abs-def*]  
**unfolding**  $rbt\text{-comp-ins ..}$

**lemma**  $rbt\text{-comp-insert}$ :  $rbt\text{-comp-insert} = ord.rbt\text{-insert } (lt\text{-of-comp } c)$   
**unfolding**  $rbt\text{-comp-insert-def}$ [*abs-def*]  $ord.rbt\text{-insert-def}$ [*abs-def*]  
**unfolding**  $rbt\text{-comp-insert-with-key ..}$

**lemma**  $rbt\text{-comp-del}$ :  $rbt\text{-comp-del} = ord.rbt\text{-del } (lt\text{-of-comp } c)$   
**proof** – {

**fix**  $k a b$  **and**  $s t :: ('a, 'b)rbt$   
**have**  
 $rbt\text{-comp-del-from-left } k t a b s = ord.rbt\text{-del-from-left } (lt\text{-of-comp } c) k t a b s$   
 $rbt\text{-comp-del-from-right } k t a b s = ord.rbt\text{-del-from-right } (lt\text{-of-comp } c) k t a b$

$s$

$rbt\text{-comp-del } k t = ord.rbt\text{-del } (lt\text{-of-comp } c) k t$

**by** (*induct*  $k t a b s$  **and**  $k t a b s$  **and**  $k t$  *rule*:  $rbt\text{-comp-del-from-left-rbt-comp-del-from-right-rbt-comp-del.induct}$   
*unfold*

$rbt\text{-comp-del.simps } ord.rbt\text{-del.simps}$   
 $rbt\text{-comp-del-from-left.simps } ord.rbt\text{-del-from-left.simps}$   
 $rbt\text{-comp-del-from-right.simps } ord.rbt\text{-del-from-right.simps}$   
*comparator.two-comparisons-into-case-order*[ $OF c$ ],  
*auto split: order.split*)

}

**thus** *?thesis* **by** (*intro ext*)

**qed**

**lemma**  $rbt\text{-comp-delete}$ :  $rbt\text{-comp-delete} = ord.rbt\text{-delete } (lt\text{-of-comp } c)$   
**unfolding**  $rbt\text{-comp-delete-def}$ [*abs-def*]  $ord.rbt\text{-delete-def}$ [*abs-def*]  
**unfolding**  $rbt\text{-comp-del ..}$

**lemma**  $rbt\text{-comp-bulkload}$ :  $rbt\text{-comp-bulkload} = ord.rbt\text{-bulkload } (lt\text{-of-comp } c)$   
**unfolding**  $rbt\text{-comp-bulkload-def}$ [*abs-def*]  $ord.rbt\text{-bulkload-def}$ [*abs-def*]  
**unfolding**  $rbt\text{-comp-insert ..}$

**lemma**  $rbt\text{-comp-map-entry}$ :  $rbt\text{-comp-map-entry} = ord.rbt\text{-map-entry } (lt\text{-of-comp } c)$   
**proof** (*intro ext*)

**fix**  $f k$  **and**  $t :: ('a, 'b)rbt$

**show**  $rbt\text{-comp-map-entry } f k t = ord.rbt\text{-map-entry } (lt\text{-of-comp } c) f k t$

**by** (*induct*  $t$ , *unfold*  $rbt\text{-comp-map-entry.simps } ord.rbt\text{-map-entry.simps}$ )

```

    comparator.two-comparisons-into-case-order[OF c]
    (auto split: order.splits)
qed

lemma comp-sunion-with: comp-sunion-with = ord.sunion-with (lt-of-comp c)
proof (intro ext)
  fix f and as bs :: ('a × 'b)list
  show comp-sunion-with f as bs = ord.sunion-with (lt-of-comp c) f as bs
  by (induct f as bs rule: comp-sunion-with.induct,
      unfold comp-sunion-with.simps ord.sunion-with.simps
      comparator.two-comparisons-into-case-order[OF c]
      (auto split: order.splits))
qed

lemma anti-sym: lt-of-comp c a x  $\implies$  lt-of-comp c x a  $\implies$  False
  by (metis c comparator.Gt-lt-conv comparator.Lt-lt-conv order.distinct(5))

lemma rbt-split-comp: rbt-split-comp t x = ord.rbt-split (lt-of-comp c) t x
  by (induction t x rule: rbt-split-comp.induct)
  (auto simp: ord.rbt-split.simps comparator.le-lt-convs[OF c]
      split: order.splits prod.splits dest: anti-sym)

lemma comp-union-with-key: rbt-comp-union-rec f t1 t2 = ord.rbt-union-rec (lt-of-comp c) f t1 t2
proof (induction f t1 t2 rule: rbt-comp-union-rec.induct)
  case (1 f t1 t2)
  obtain f' t1' t2' where flip: (f', t2', t1') =
    (if flip-rbt t2 t1 then ( $\lambda$ k v v'. f k v' v, t1, t2) else (f, t2, t1))
  by fastforce
  show ?case
  proof (cases t1')
    case (Branch - l1 a b r1)
    have t1-not-Empty: t1'  $\neq$  RBT-Impl.Empty
    by (auto simp: Branch)
    obtain l2  $\beta$  r2 where split: rbt-split-comp t2' a = (l2,  $\beta$ , r2)
    by (cases rbt-split-comp t2' a) auto
    show ?thesis
    using 1[OF flip refl - - Branch]
    unfolding rbt-comp-union-rec.simps[of - t1] ord.rbt-union-rec.simps[of - - t1]
    flip[symmetric]
    by (auto simp: Branch split rbt-split-comp[symmetric] rbt-comp-insert-with-key
        split: prod.splits)
  qed (auto simp: rbt-comp-union-rec.simps[of - t1] ord.rbt-union-rec.simps[of - - t1]
      flip[symmetric]
      rbt-comp-insert-with-key rbt-split-comp[symmetric])
qed

lemma comp-sinter-with: comp-sinter-with = ord.sinter-with (lt-of-comp c)
proof (intro ext)

```



```

fix f and as bs :: ('a × 'b)list
show comp-sinter-with f as bs = ord.sinter-with (lt-of-comp c) f as bs
  by (induct f as bs rule: comp-sinter-with.induct,
      unfold comp-sinter-with.simps ord.sinter-with.simps
      comparator.two-comparisons-into-case-order[OF c]
      (auto split: order.splits)
qed

lemma rbt-comp-union-with-key: rbt-comp-union-with-key = ord.rbt-union-with-key
(lt-of-comp c)
  by (rule ext)+
      (auto simp: rbt-comp-union-with-key-def rbt-comp-union-swap-rec ord.rbt-union-with-key-def
      ord.rbt-union-swap-rec comp-union-with-key)

lemma comp-inter-with-key: rbt-comp-inter-rec f t1 t2 = ord.rbt-inter-rec (lt-of-comp
c) f t1 t2
proof (induction f t1 t2 rule: rbt-comp-inter-rec.induct)
  case (1 f t1 t2)
  obtain f' t1' t2' where flip: (f', t2', t1') =
      (if flip-rbt t2 t1 then (λk v v'. f k v' v, t1, t2) else (f, t2, t1))
  by fastforce
show ?case
proof (cases t1')
  case (Branch - l1 a b r1)
  have t1'-not-Empty: t1' ≠ RBT-Impl.Empty
  by (auto simp: Branch)
  obtain l2 β r2 where split: rbt-split-comp t2' a = (l2, β, r2)
  by (cases rbt-split-comp t2' a) auto
show ?thesis
  using 1[OF flip refl - - Branch]
  unfolding rbt-comp-inter-rec.simps[of - t1] ord.rbt-inter-rec.simps[of - - t1]
flip[symmetric]
  by (auto simp: Branch split rbt-split-comp[symmetric] rbt-comp-lookup
      ord.map-filter-inter-def map-filter-comp-inter-def split: prod.splits)
qed (auto simp: rbt-comp-inter-rec.simps[of - t1] ord.rbt-inter-rec.simps[of - - t1]
flip[symmetric]
      ord.map-filter-inter-def map-filter-comp-inter-def rbt-comp-lookup rbt-split-comp[symmetric])
qed

lemma rbt-comp-inter-with-key: rbt-comp-inter-with-key = ord.rbt-inter-with-key
(lt-of-comp c)
  by (rule ext)+
      (auto simp: rbt-comp-inter-with-key-def rbt-comp-inter-swap-rec
      ord.rbt-inter-with-key-def ord.rbt-inter-swap-rec comp-inter-with-key)

lemma comp-minus: comp-minus t1 t2 = ord.rbt-minus-rec (lt-of-comp c) t1 t2
proof (induction t1 t2 rule: comp-minus.induct)
  case (1 t1 t2)
  show ?case

```

```

proof (cases t2)
  case (Branch - l2 a u r2)
  have t2-not-Empty: t2 ≠ RBT-Impl.Empty
    by (auto simp: Branch)
  obtain l1 β r1 where split: rbt-split-comp t1 a = (l1, β, r1)
    by (cases rbt-split-comp t1 a) auto
  show ?thesis
    using 1[OF - - Branch]
    unfolding comp-minus.simps[of t1 t2] ord.rbt-minus-rec.simps[of - t1 t2]
  by (auto simp: Branch split rbt-split-comp[symmetric] rbt-comp-delete rbt-comp-lookup
    filter-comp-minus-def ord.filter-minus-def split: prod.splits)
qed (auto simp: comp-minus.simps[of t1] ord.rbt-minus-rec.simps[of - t1]
  filter-comp-minus-def ord.filter-minus-def
  rbt-comp-delete rbt-comp-lookup rbt-split-comp[symmetric])
qed

```

```

lemma rbt-comp-minus: rbt-comp-minus = ord.rbt-minus (lt-of-comp c)
  by (rule ext)+ (auto simp: rbt-comp-minus-def ord.rbt-minus-def comp-minus)

```

```

lemmas rbt-comp-simps =
  rbt-comp-insert
  rbt-comp-lookup
  rbt-comp-delete
  rbt-comp-bulkload
  rbt-comp-map-entry
  rbt-comp-union-with-key
  rbt-comp-inter-with-key
  rbt-comp-minus

```

```

end

```

```

end

```

```

end

```

## 4 Generating Comparators

```

theory Comparator-Generator

```

```

imports

```

```

  ../Generator-Aux

```

```

  ../Derive-Manager

```

```

  Comparator

```

```

begin

```

```

typedecl ('a,'b,'c,'z)type

```

In the following, we define a generator which for a given datatype ('a, 'b, 'c, 'z) *Comparator-Generator.type* constructs a comparator of type 'a comparator ⇒ 'b comparator ⇒ 'c comparator ⇒ 'z comparator ⇒ ('a, 'b, 'c, 'z) *Comparator-Generator.type*. To this end, we first compare the index of the constructors, then for equal constructors, we compare the arguments

recursively and combine the results lexicographically.

**hide-type** *type*

#### 4.1 Lexicographic combination of *order*

**fun** *comp-lex* :: *order list*  $\Rightarrow$  *order*

**where**

*comp-lex* (*c* # *cs*) = (*case c of Eq*  $\Rightarrow$  *comp-lex cs* | -  $\Rightarrow$  *c*) |  
*comp-lex* [] = *Eq*

#### 4.2 Improved code for non-lazy languages

The following equations will eliminate all occurrences of *comp-lex* in the generated code of the comparators.

**lemma** *comp-lex-unfolds*:

*comp-lex* [] = *Eq*  
*comp-lex* [*c*] = *c*  
*comp-lex* (*c* # *d* # *cs*) = (*case c of Eq*  $\Rightarrow$  *comp-lex (d # cs)* | *z*  $\Rightarrow$  *z*)  
**by** (*cases c, auto*)<sup>+</sup>

#### 4.3 Pointwise properties for equality, symmetry, and transitivity

The pointwise properties are important during inductive proofs of soundness of comparators. They are defined in a way that are combinable with *comp-lex*.

**lemma** *comp-lex-eq*: *comp-lex os* = *Eq*  $\longleftrightarrow$  ( $\forall$  *ord*  $\in$  *set os*. *ord* = *Eq*)

**by** (*induct os*) (*auto split: order.splits*)

**definition** *trans-order* :: *order*  $\Rightarrow$  *order*  $\Rightarrow$  *bool* **where**

*trans-order* *x y z*  $\longleftrightarrow$  *x*  $\neq$  *Gt*  $\longrightarrow$  *y*  $\neq$  *Gt*  $\longrightarrow$  *z*  $\neq$  *Gt*  $\wedge$  ((*x* = *Lt*  $\vee$  *y* = *Lt*)  $\longrightarrow$  *z* = *Lt*)

**lemma** *trans-orderI*:

(*x*  $\neq$  *Gt*  $\implies$  *y*  $\neq$  *Gt*  $\implies$  *z*  $\neq$  *Gt*  $\wedge$  ((*x* = *Lt*  $\vee$  *y* = *Lt*)  $\longrightarrow$  *z* = *Lt*))  $\implies$   
*trans-order* *x y z*

**by** (*simp add: trans-order-def*)

**lemma** *trans-orderD*:

**assumes** *trans-order* *x y z* **and** *x*  $\neq$  *Gt* **and** *y*  $\neq$  *Gt*  
**shows** *z*  $\neq$  *Gt* **and** *x* = *Lt*  $\vee$  *y* = *Lt*  $\implies$  *z* = *Lt*  
**using** *assms* **by** (*auto simp: trans-order-def*)

**lemma** *All-less-Suc*:

( $\forall$  *i* < *Suc x*. *P i*)  $\longleftrightarrow$  *P 0*  $\wedge$  ( $\forall$  *i* < *x*. *P (Suc i)*)  
**using** *less-Suc-eq-0-disj* **by** *force*

**lemma** *comp-lex-trans*:

```

assumes length xs = length ys
and length ys = length zs
and  $\forall i < \text{length } zs. \text{trans-order } (xs ! i) (ys ! i) (zs ! i)$ 
shows trans-order (comp-lex xs) (comp-lex ys) (comp-lex zs)
using assms
proof (induct xs ys zs rule: list-induct3)
case (Cons x xs y ys z zs)
then show ?case
by (intro trans-orderI)
      (cases x y z rule: order.exhaust [case-product order.exhaust order.exhaust],
       auto simp: All-less-Suc dest: trans-orderD)
qed (simp add: trans-order-def)

```

```

lemma comp-lex-sym:
assumes length xs = length ys
and  $\forall i < \text{length } ys. \text{invert-order } (xs ! i) = ys ! i$ 
shows invert-order (comp-lex xs) = comp-lex ys
using assms by (induct xs ys rule: list-induct2, simp, case-tac x) fastforce+

```

```

declare comp-lex.simps [simp del]

```

```

definition peq-comp :: 'a comparator  $\Rightarrow$  'a  $\Rightarrow$  bool
where
  peq-comp acomp x  $\longleftrightarrow$  ( $\forall y. \text{acomp } x y = \text{Eq} \longleftrightarrow x = y$ )

```

```

lemma peq-compD: peq-comp acomp x  $\Longrightarrow$  acomp x y = Eq  $\longleftrightarrow$  x = y
unfolding peq-comp-def by auto

```

```

lemma peq-compI: ( $\bigwedge y. \text{acomp } x y = \text{Eq} \longleftrightarrow x = y$ )  $\Longrightarrow$  peq-comp acomp x
unfolding peq-comp-def by auto

```

```

definition psym-comp :: 'a comparator  $\Rightarrow$  'a  $\Rightarrow$  bool where
  psym-comp acomp x  $\longleftrightarrow$  ( $\forall y. \text{invert-order } (\text{acomp } x y) = (\text{acomp } y x)$ )

```

```

lemma psym-compD:
assumes psym-comp acomp x
shows invert-order (acomp x y) = (acomp y x)
using assms unfolding psym-comp-def by blast+

```

```

lemma psym-compI:
assumes  $\bigwedge y. \text{invert-order } (\text{acomp } x y) = (\text{acomp } y x)$ 
shows psym-comp acomp x
using assms unfolding psym-comp-def by blast

```

```

definition ptrans-comp :: 'a comparator  $\Rightarrow$  'a  $\Rightarrow$  bool where
  ptrans-comp acomp x  $\longleftrightarrow$  ( $\forall y z. \text{trans-order } (\text{acomp } x y) (\text{acomp } y z) (\text{acomp } x z)$ )

```

**lemma** *ptrans-compD*:  
**assumes** *ptrans-comp acomp x*  
**shows** *trans-order (acompe x y) (acompe y z) (acompe x z)*  
**using** *assms unfolding ptrans-comp-def by blast+*

**lemma** *ptrans-compI*:  
**assumes**  $\bigwedge y z. \text{trans-order } (acompe x y) (acompe y z) (acompe x z)$   
**shows** *ptrans-comp acomp x*  
**using** *assms unfolding ptrans-comp-def by blast*

#### 4.4 Separate properties of comparators

**definition** *eq-comp* :: 'a comparator  $\Rightarrow$  bool **where**  
*eq-comp acomp*  $\longleftrightarrow (\forall x. \text{peq-comp } acomp x)$

**lemma** *eq-compD2*: *eq-comp acomp*  $\Longrightarrow$  *peq-comp acomp x*  
**unfolding** *eq-comp-def by blast*

**lemma** *eq-compI2*:  $(\bigwedge x. \text{peq-comp } acomp x)$   $\Longrightarrow$  *eq-comp acomp*  
**unfolding** *eq-comp-def by blast*

**definition** *trans-comp* :: 'a comparator  $\Rightarrow$  bool **where**  
*trans-comp acomp*  $\longleftrightarrow (\forall x. \text{ptrans-comp } acomp x)$

**lemma** *trans-compD2*: *trans-comp acomp*  $\Longrightarrow$  *ptrans-comp acomp x*  
**unfolding** *trans-comp-def by blast*

**lemma** *trans-compI2*:  $(\bigwedge x. \text{ptrans-comp } acomp x)$   $\Longrightarrow$  *trans-comp acomp*  
**unfolding** *trans-comp-def by blast*

**definition** *sym-comp* :: 'a comparator  $\Rightarrow$  bool **where**  
*sym-comp acomp*  $\longleftrightarrow (\forall x. \text{psym-comp } acomp x)$

**lemma** *sym-compD2*:  
*sym-comp acomp*  $\Longrightarrow$  *psym-comp acomp x*  
**unfolding** *sym-comp-def by blast*

**lemma** *sym-compI2*:  $(\bigwedge x. \text{psym-comp } acomp x)$   $\Longrightarrow$  *sym-comp acomp*  
**unfolding** *sym-comp-def by blast*

**lemma** *eq-compD*: *eq-comp acomp*  $\Longrightarrow$  *acompe x y = Eq*  $\longleftrightarrow$  *x = y*  
**by** (*rule peq-compD[OF eq-compD2]*)

**lemma** *eq-compI*:  $(\bigwedge x y. \text{acompe } x y = Eq \longleftrightarrow x = y)$   $\Longrightarrow$  *eq-comp acomp*  
**by** (*intro eq-compI2 peq-compI*)

**lemma** *trans-compD*: *trans-comp acomp*  $\Longrightarrow$  *trans-order (acompe x y) (acompe y z)*  
*(acompe x z)*

```

by (rule ptrans-compD[OF trans-compD2])

lemma trans-compI: ( $\bigwedge x y z. \text{trans-order } (acom\ x\ y)\ (acom\ y\ z)\ (acom\ x\ z)$ )
 $\implies$  trans-comp acom
  by (intro trans-compI2 ptrans-compI)

lemma sym-compD:
  sym-comp acom  $\implies$  invert-order (acom x y) = (acom y x)
  by (rule psym-compD[OF sym-compD2])

lemma sym-compI: ( $\bigwedge x y. \text{invert-order } (acom\ x\ y) = (acom\ y\ x)$ )  $\implies$  sym-comp
acom
  by (intro sym-compI2 psym-compI)

lemma eq-sym-trans-imp-comparator:
  assumes eq-comp acom and sym-comp acom and trans-comp acom
  shows comparator acom
proof
  fix x y z
  show invert-order (acom x y) = acom y x
    using sym-compD [OF  $\langle$ sym-comp acom $\rangle$ ] .
  {
    assume acom x y = Eq
    with eq-compD [OF  $\langle$ eq-comp acom $\rangle$ ]
    show x = y by blast
  }
  {
    assume acom x y = Lt and acom y z = Lt
    with trans-orderD [OF trans-compD [OF  $\langle$ trans-comp acom $\rangle$ ], of x y z]
    show acom x z = Lt by auto
  }
qed

lemma comparator-imp-eq-sym-trans:
  assumes comparator acom
  shows eq-comp acom sym-comp acom trans-comp acom
proof -
  interpret comparator acom by fact
  show eq-comp acom using eq by (intro eq-compI, auto)
  show sym-comp acom using sym by (intro sym-compI, auto)
  show trans-comp acom
  proof (intro trans-compI trans-orderI)
    fix x y z
    assume acom x y  $\neq$  Gt acom y z  $\neq$  Gt
    thus acom x z  $\neq$  Gt  $\wedge$  (acom x y = Lt  $\vee$  acom y z = Lt  $\longrightarrow$  acom x z =
Lt)
      using comp-trans [of x y z] and eq [of x y] and eq [of y z]
      by (cases acom x y acom y z rule: order.exhaust [case-product order.exhaust])
  auto

```

```

qed
qed

context
  fixes acomp :: 'a comparator
  assumes c: comparator acomp
begin
lemma comp-to-psym-comp: psym-comp acomp x
  using comparator-imp-eq-sym-trans[OF c]
  by (intro sym-compD2)

lemma comp-to-peq-comp: peq-comp acomp x
  using comparator-imp-eq-sym-trans [OF c]
  by (intro eq-compD2)

lemma comp-to-ptrans-comp: ptrans-comp acomp x
  using comparator-imp-eq-sym-trans [OF c]
  by (intro trans-compD2)
end

```

#### 4.5 Auxiliary Lemmas for Comparator Generator

```

lemma forall-finite: (∀ i < (0 :: nat). P i) = True
  (∀ i < Suc 0. P i) = P 0
  (∀ i < Suc (Suc x). P i) = (P 0 ∧ (∀ i < Suc x. P (Suc i)))
  by (auto, case-tac i, auto)

```

```

lemma trans-order-different:
  trans-order a b Lt
  trans-order Gt b c
  trans-order a Gt c
  by (intro trans-orderI, auto)+

```

```

lemma length-nth-simps:
  length [] = 0 length (x # xs) = Suc (length xs)
  (x # xs) ! 0 = x (x # xs) ! (Suc n) = xs ! n by auto

```

#### 4.6 The Comparator Generator

```
ML-file <comparator-generator.ML>
```

```
end
```

#### 4.7 Compare Generator

```

theory Compare-Generator
imports
  Comparator-Generator
  Compare
begin

```

We provide a generator which takes the comparators of the comparator generator to synthesize suitable *compare*-functions from the *compare*-class.

One can further also use these comparison functions to derive an instance of the *compare-order*-class, and therefore also for *linorder*. In total, we provide the three *derive*-methods where the example type *prod* can be replaced by any other datatype.

- *derive compare prod* creates an instance *prod* :: (*compare*, *compare*) *compare*.
- *derive compare-order prod* creates an instance *prod* :: (*compare*, *compare*) *compare-order*.
- *derive linorder prod* creates an instance *prod* :: (*linorder*, *linorder*) *linorder*.

Usually, the use of *derive linorder* is not recommended if there are comparators available: Internally, the linear orders will directly be converted into comparators, so a direct use of the comparators will result in more efficient generated code. This command is mainly provided as a convenience method where comparators are not yet present. For example, at the time of writing, the Container Framework has partly been adapted to internally use comparators, whereas in other AFP-entries, we did not integrate comparators.

**lemma** *linorder-axiomsD*: **assumes** *class.linorder le lt*

**shows**

*lt x y = (le x y ∧ ¬ le y x) (is ?a)*

*le x x (is ?b)*

*le x y ⇒ le y z ⇒ le x z (is ?c1 ⇒ ?c2 ⇒ ?c3)*

*le x y ⇒ le y x ⇒ x = y (is ?d1 ⇒ ?d2 ⇒ ?d3)*

*le x y ∨ le y x (is ?e)*

**proof** –

**interpret** *linorder le lt* **by** *fact*

**show** *?a ?b ?c1 ⇒ ?c2 ⇒ ?c3 ?d1 ⇒ ?d2 ⇒ ?d3 ?e* **by** *auto*

**qed**

**named-theorems** *compare-simps simp theorems to derive compare = comparator-of*

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

**end**

## 4.8 Defining Comparators and Compare-Instances for Common Types

**theory** *Compare-Instances*

**imports**



*Compare-Generator*  
*HOL-Library.Char-ord*

**begin**

For all of the following types, we define comparators and register them in the class *compare*: *int*, *integer*, *nat*, *char*, *bool*, *unit*, *sum*, *option*, *list*, and *prod*. We do not register those classes in *compare-order* where so far no linear order is defined, in particular if there are conflicting orders, like pair-wise or lexicographic comparison on pairs.

For *int*, *nat*, *integer* and *char* we just use their linear orders as comparators.

**derive** (*linorder*) *compare-order int integer nat char*

For *sum*, *list*, *prod*, and *option* we generate comparators which are however are not used to instantiate *linorder*.

**derive** *compare sum list prod option*

We do not use the linear order to define the comparator for *bool* and *unit*, but implement more efficient ones.

**fun** *comparator-unit* :: *unit comparator* **where**  
*comparator-unit x y = Eq*

**fun** *comparator-bool* :: *bool comparator* **where**  
*comparator-bool False False = Eq*  
| *comparator-bool False True = Lt*  
| *comparator-bool True True = Eq*  
| *comparator-bool True False = Gt*

**lemma** *comparator-unit: comparator comparator-unit*  
**by** (*unfold-locales, auto*)

**lemma** *comparator-bool: comparator comparator-bool*

**proof**

**fix** *x y z* :: *bool*  
**show** *invert-order (comparator-bool x y) = comparator-bool y x* **by** (*cases x, (cases y, auto)+*)  
**show** *comparator-bool x y = Eq*  $\implies$  *x = y* **by** (*cases x, (cases y, auto)+*)  
**show** *comparator-bool x y = Lt*  $\implies$  *comparator-bool y z = Lt*  $\implies$  *comparator-bool x z = Lt*  
**by** (*cases x, (cases y, auto), cases y, (cases z, auto)+*)

**qed**

**local-setup**  $\langle$

*Comparator-Generator.register-foreign-comparator* @*{typ unit}*  
@*{term comparator-unit}*  
@*{thm comparator-unit}*

$\rangle$

```

local-setup <
  Comparator-Generator.register-foreign-comparator @{typ bool}
    @{term comparator-bool}
    @{thm comparator-bool}
  >

```

```

derive compare bool unit

```

It is not directly possible to *derive* (*linorder*) *bool unit*, since *compare* was not defined as *comparator-of*, but as *comparator-bool*. However, we can manually prove this equivalence and then use this knowledge to prove the instance of *compare-order*.

```

lemma comparator-bool-comparator-of [compare-simps]:

```

```

  comparator-bool = comparator-of

```

```

proof (intro ext)

```

```

  fix a b

```

```

  show comparator-bool a b = comparator-of a b

```

```

    unfolding comparator-of-def

```

```

    by (cases a, (cases b, auto))

```

```

qed

```

```

lemma comparator-unit-comparator-of [compare-simps]:

```

```

  comparator-unit = comparator-of

```

```

proof (intro ext)

```

```

  fix a b

```

```

  show comparator-unit a b = comparator-of a b

```

```

    unfolding comparator-of-def by auto

```

```

qed

```

```

derive (linorder) compare-order bool unit

```

```

end

```

## 4.9 Defining Compare-Order-Instances for Common Types

```

theory Compare-Order-Instances

```

```

imports

```

```

  Compare-Instances

```

```

  HOL-Library.List-Lexorder

```

```

  HOL-Library.Product-Lexorder

```

```

  HOL-Library.Option-ord

```

```

begin

```

We now also instantiate class *compare-order* and not only *compare*. Here, we also prove that our definitions do not clash with existing orders on *list*, *option*, and *prod*.

For *sum* we just define the linear orders via their comparator.

```

derive compare-order sum

```

```

instance list :: (compare-order)compare-order
proof
  note [simp] = le-of-comp-def lt-of-comp-def comparator-of-def
  show le-of-comp (compare :: 'a list comparator) = ( $\leq$ )
    unfolding compare-list-def compare-is-comparator-of
  proof (intro ext)
    fix xs ys :: 'a list
    show le-of-comp (comparator-list comparator-of) xs ys = (xs  $\leq$  ys)
    proof (induct xs arbitrary: ys)
      case (Nil ys)
      show ?case
      by (cases ys, simp-all)
    next
      case (Cons x xs yys) note IH = this
      thus ?case
      proof (cases yys)
        case Nil
        thus ?thesis by auto
      next
        case (Cons y ys)
        show ?thesis unfolding Cons
          using IH[of ys]
          by (cases x y rule: linorder-cases, auto)
      qed
    qed
  qed
  show lt-of-comp (compare :: 'a list comparator) = ( $<$ )
    unfolding compare-list-def compare-is-comparator-of
  proof (intro ext)
    fix xs ys :: 'a list
    show lt-of-comp (comparator-list comparator-of) xs ys = (xs  $<$  ys)
    proof (induct xs arbitrary: ys)
      case (Nil ys)
      show ?case
      by (cases ys, simp-all)
    next
      case (Cons x xs yys) note IH = this
      thus ?case
      proof (cases yys)
        case Nil
        thus ?thesis by auto
      next
        case (Cons y ys)
        show ?thesis unfolding Cons
          using IH[of ys]
          by (cases x y rule: linorder-cases, auto)
      qed
    qed
  qed

```

qed

**instance** prod :: (compare-order, compare-order)compare-order

**proof**

**note** [simp] = le-of-comp-def lt-of-comp-def comparator-of-def

**show** le-of-comp (compare :: ('a,'b)prod comparator) = ( $\leq$ )

**unfolding** compare-prod-def compare-is-comparator-of

**proof** (intro ext)

**fix** xy1 xy2 :: ('a,'b)prod

**show** le-of-comp (comparator-prod comparator-of comparator-of) xy1 xy2 =  
(xy1  $\leq$  xy2)

**by** (cases xy1, cases xy2, auto)

qed

**show** lt-of-comp (compare :: ('a,'b)prod comparator) = ( $<$ )

**unfolding** compare-prod-def compare-is-comparator-of

**proof** (intro ext)

**fix** xy1 xy2 :: ('a,'b)prod

**show** lt-of-comp (comparator-prod comparator-of comparator-of) xy1 xy2 = (xy1  
 $<$  xy2)

**by** (cases xy1, cases xy2, auto)

qed

qed

**instance** option :: (compare-order)compare-order

**proof**

**note** [simp] = le-of-comp-def lt-of-comp-def comparator-of-def

**show** le-of-comp (compare :: 'a option comparator) = ( $\leq$ )

**unfolding** compare-option-def compare-is-comparator-of

**proof** (intro ext)

**fix** xy1 xy2 :: 'a option

**show** le-of-comp (comparator-option comparator-of) xy1 xy2 = (xy1  $\leq$  xy2)

**by** (cases xy1, (cases xy2, auto split: if-splits)+)

qed

**show** lt-of-comp (compare :: 'a option comparator) = ( $<$ )

**unfolding** compare-option-def compare-is-comparator-of

**proof** (intro ext)

**fix** xy1 xy2 :: 'a option

**show** lt-of-comp (comparator-option comparator-of) xy1 xy2 = (xy1  $<$  xy2)

**by** (cases xy1, (cases xy2, auto split: if-splits)+)

qed

qed

end

## 4.10 Compare Instance for Rational Numbers

**theory** Compare-Rat

**imports**

Compare-Generator

```

    HOL.Rat
begin

derive (linorder) compare-order rat

end

```

#### 4.11 Compare Instance for Real Numbers

```

theory Compare-Real
imports
  Compare-Generator
  HOL.Real
begin

derive (linorder) compare-order real

lemma invert-order-compare-real[simp]:  $\bigwedge x y :: real. invert-order (compare x y) = compare y x$ 
  by (simp add: comparator-of-def compare-is-comparator-of)

end

```

### 5 Checking Equality Without "="

```

theory Equality-Generator
imports
  ../Generator-Aux
  ../Derive-Manager
begin

typedecl ('a,'b,'c,'z)type

```

In the following, we define a generator which for a given datatype  $( 'a, 'b, 'c, 'z )$  *Equality-Generator.type* constructs an equality-test function of type  $( 'a \Rightarrow 'a \Rightarrow bool ) \Rightarrow ( 'b \Rightarrow 'b \Rightarrow bool ) \Rightarrow ( 'c \Rightarrow 'c \Rightarrow bool ) \Rightarrow ( 'z \Rightarrow 'z \Rightarrow bool ) \Rightarrow ( 'a, 'b, 'c, 'z )$  *Equality-Generator.type*  $\Rightarrow ( 'a, 'b, 'c, 'z )$  *Equality-Generator.type*  $\Rightarrow bool$ . These functions are essential to synthesize conditional equality functions in the container framework, where a strict membership in the *equal*-class must not be enforced.

```
hide-type type
```

Just a constant to define conjunction on lists of booleans, which will be used to merge the results when having compared the arguments of identical constructors.

```

definition list-all-eq :: bool list  $\Rightarrow$  bool where
  list-all-eq = list-all id

```

## 5.1 Improved Code for Non-Lazy Languages

The following equations will eliminate all occurrences of *list-all-eq* in the generated code of the equality functions.

**lemma** *list-all-eq-unfold*:

*list-all-eq* [] = True

*list-all-eq* [b] = b

*list-all-eq* (b1 # b2 # bs) = (b1 ∧ *list-all-eq* (b2 # bs))

**unfolding** *list-all-eq-def*

**by** *auto*

**lemma** *list-all-eq*: *list-all-eq* bs  $\longleftrightarrow$  ( $\forall b \in \text{set } bs. b$ )

**unfolding** *list-all-eq-def list-all-iff* **by** *auto*

## 5.2 Partial Equality Property

We require a partial property which can be used in inductive proofs.

**type-synonym** 'a *equality* = 'a  $\Rightarrow$  'a  $\Rightarrow$  bool

**definition** *pequality* :: 'a *equality*  $\Rightarrow$  'a  $\Rightarrow$  bool

**where**

*pequality* aeq x  $\longleftrightarrow$  ( $\forall y. \text{aeq } x \ y \longleftrightarrow x = y$ )

**lemma** *pequalityD*: *pequality* aeq x  $\Longrightarrow$  aeq x y  $\longleftrightarrow$  x = y

**unfolding** *pequality-def* **by** *auto*

**lemma** *pequalityI*: ( $\bigwedge y. \text{aeq } x \ y \longleftrightarrow x = y$ )  $\Longrightarrow$  *pequality* aeq x

**unfolding** *pequality-def* **by** *auto*

## 5.3 Global equality property

**definition** *equality* :: 'a *equality*  $\Rightarrow$  bool **where**

*equality* aeq  $\longleftrightarrow$  ( $\forall x. \text{pequality } \text{aeq } x$ )

**lemma** *equalityD2*: *equality* aeq  $\Longrightarrow$  *pequality* aeq x

**unfolding** *equality-def* **by** *blast*

**lemma** *equalityI2*: ( $\bigwedge x. \text{pequality } \text{aeq } x$ )  $\Longrightarrow$  *equality* aeq

**unfolding** *equality-def* **by** *blast*

**lemma** *equalityD*: *equality* aeq  $\Longrightarrow$  aeq x y  $\longleftrightarrow$  x = y

**by** (*rule pequalityD[OF equalityD2]*)

**lemma** *equalityI*: ( $\bigwedge x \ y. \text{aeq } x \ y \longleftrightarrow x = y$ )  $\Longrightarrow$  *equality* aeq

**by** (*intro equalityI2 pequalityI*)

**lemma** *equality-imp-eq*:

*equality* aeq  $\Longrightarrow$  aeq = (=)

**by** (*intro ext, auto dest: equalityD*)

```

lemma eq-equality: equality (=)
  by (rule equalityI, simp)

lemma equality-def': equality f = (f = (=))
  using equality-imp-eq eq-equality by blast

```

## 5.4 The Generator

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

**hide-fact** (**open**) *equalityI*

**end**

## 5.5 Defining Equality-Functions for Common Types

**theory** *Equality-Instances*

**imports**

*Equality-Generator*

**begin**

For all of the following types, we register equality-functions. *int*, *integer*, *nat*, *char*, *bool*, *unit*, *sum*, *option*, *list*, and *prod*. For types without type parameters, we use plain (=), and for the others we use generated ones. These functions will be essential, when the generator is later on invoked on types, which in their definition use one these types.

**derive** (*eq*) *equality int integer nat char bool unit*

**derive** *equality sum list prod option*

**end**

## 6 Generating Hash-Functions

**theory** *Hash-Generator*

**imports**

*../Generator-Aux*

*../Derive-Manager*

*Collections.HashCode*

**begin**

As usual, in the generator we use a dedicated function to combine the results from evaluating the hash-function of the arguments of a constructor, to deliver the global hash-value.

**fun** *hash-combine* :: *hashcode list*  $\Rightarrow$  *hashcode list*  $\Rightarrow$  *hashcode* **where**

*hash-combine* [] [x] = x

| *hash-combine* (y # ys) (z # zs) = y \* z + *hash-combine* ys zs

| *hash-combine* - - = 0

The first argument of *hash-combine* originates from evaluating the hash-function on the arguments of a constructor, and the second argument of *hash-combine* will be static *magic* numbers which are generated within the generator.

## 6.1 Improved Code for Non-Lazy Languages

```
lemma hash-combine-unfold:
  hash-combine [] [x] = x
  hash-combine (y # ys) (z # zs) = y * z + hash-combine ys zs
  by auto
```

## 6.2 The Generator

ML-file `<hash-generator.ML>`

end

## 6.3 Defining Hash-Functions for Common Types

```
theory Hash-Instances
imports
  Hash-Generator
begin
```

For all of the following types, we register hashcode-functions. *int*, *integer*, *nat*, *char*, *bool*, *unit*, *sum*, *option*, *list*, and *prod*. For types without type parameters, we use plain *hashcode*, and for the others we use generated ones.

```
derive (hashcode) hash-code int integer bool char unit nat
```

```
derive hash-code prod sum option list
```

There is no need to *derive hashable prod sum option list* since all of these types are already instances of class *hashable*. Still the above command is necessary to register these types in the generator.

end

## 7 Countable Datatypes

```
theory Countable-Generator
imports
  HOL-Library.Countable
  ../Derive-Manager
begin
```

Brian Huffman and Alexander Krauss (old datatype), and Jasmin Blanchette (BNF datatype) have developed tactics which automatically can prove that



a datatype is countable. We just make this tactic available in the derive-manager so that one can conveniently write `derive countable some-datatype`.

## 7.1 Installing the tactic

There is nothing more to do, then to write some boiler-plate ML-code for class-instantiation.

```

setup <
  let
    fun derive dtyp-name - thy =
      let
        val base-name = Long-Name.base-name dtyp-name
        val - = writeln (proving that datatype ^ base-name ^ is countable)
        val sort = @{sort countable}
        val vs =
          let val i = BNF-LFP-Compat.the-spec thy dtyp-name |> #1
          in map (fn (n,-) => (n, sort)) i end
        val thy' = Class.instantiation ([dtyp-name],vs,sort) thy
          |> Class.prove-instantiation-exit (fn ctxt => countable-tac ctxt 1)
        val - = writeln (registered ^ base-name ^ in class countable)
      in thy' end
    in
      Derive-Manager.register-derive countable register datatypes is class countable
  derive
  end
  >

end

```

## 8 Loading Existing Derive-Commands

```

theory Derive
imports
  Comparator-Generator/Compare-Instances
  Equality-Generator/Equality-Instances
  Hash-Generator/Hash-Instances
  Countable-Generator/Countable-Generator
begin

```

We just load the commands to derive comparators, equality-functions, hash-functions, and the command to show that a datatype is countable, so that now all of them are available. There are further generators available in the AFP entries Containers and Show.

```

print-derives

end

```

## 9 Examples

```
theory Derive-Examples
imports
  Derive
  Comparator-Generator/Compare-Order-Instances
  Equality-Generator/Equality-Instances
  HOL.Rat
begin
```

### 9.1 Rational Numbers

The rational numbers are not a datatype, so it will not be possible to derive corresponding instances of comparators, hashcodes, etc. via the generators. But we can and should still register the existing instances, so that later datatypes are supported which use rational numbers.

Use the linear order on rationals to define the *compare-order*-instance.

```
derive (linorder) compare-order rat
```

Use (=) as equality function.

```
derive (eq) equality rat
```

First manually define a hashcode function.

```
instantiation rat :: hashable
```

```
begin
```

```
definition def-hashmap-size = ( $\lambda$ - :: rat itself. 10)
```

```
definition hashcode (r :: rat) = hashcode (quotient-of r)
```

```
instance
```

```
  by (intro-classes)(simp-all add: def-hashmap-size-rat-def)
```

```
end
```

And then register it at the generator.

```
derive (hashcode) hash-code rat
```

### 9.2 A Datatype Without Nested Recursion

```
datatype 'a bintree = BEmpty | BNode 'a bintree 'a 'a bintree
```

```
derive compare-order bintree
```

```
derive countable bintree
```

```
derive equality bintree
```

```
derive hashable bintree
```

### 9.3 Using Other datatypes

```
datatype nat-list-list = NNil | CCons nat list  $\times$  rat option nat-list-list
```

```
derive compare-order nat-list-list
```

```

derive countable nat-list-list
derive (eq) equality nat-list-list
derive hashable nat-list-list

```

## 9.4 Mutual Recursion

```

datatype
  'a mtree = MEmpty | MNode 'a 'a mtree-list and
  'a mtree-list = MNil | MCons 'a mtree 'a mtree-list

```

```

derive compare-order mtree mtree-list
derive countable mtree mtree-list
derive hashable mtree mtree-list

```

For *derive* (*equality*|*comparator*|*hash-code*) *mutual-recursive-type* there is the speciality that only one of the mutual recursive types has to be mentioned in order to register all of them. So one of *mtree* and *mtree-list* suffices.

```

derive equality mtree

```

## 9.5 Nested recursion

```

datatype 'a tree = Empty | Node 'a 'a tree list
datatype 'a ttree = TEmpty | TNode 'a 'a ttree list tree

```

```

derive compare-order tree ttree
derive countable tree ttree
derive equality tree ttree
derive hashable tree ttree

```

## 9.6 Examples from IsaFoR

```

datatype ('f,'v) term = Var 'v | Fun 'f ('f,'v) term list
datatype ('f, 'l) lab =
  Lab ('f, 'l) lab 'l
| FunLab ('f, 'l) lab ('f, 'l) lab list
| UnLab 'f
| Sharp ('f, 'l) lab

```

```

derive compare-order term lab
derive countable term lab
derive equality term lab
derive hashable term lab

```

## 9.7 A Complex Datatype

The following datatype has nested and mutual recursion, and uses other datatypes.

```

datatype ('a, 'b) complex =
  C1 nat 'a ttree × rat + ('a,'b) complex list |

```

*C2 ('a, 'b) complex list tree tree 'b ('a, 'b) complex ('a, 'b) complex2 ttree list*  
**and** (*'a, 'b) complex2 = D1 ('a, 'b) complex ttree*

On this last example type we illustrate the difference of the various comparator- and order-generators.

For *complex* we create an instance of *compare-order* which also defines a linear order. Note however that the instance will be *complex :: (compare, compare) compare-order*, i.e., the argument types have to be in class *compare*.

For *complex2* we only derive *compare* which is not a subclass of *linorder*. The instance will be *complex2 :: (compare, compare) compare*, i.e., again the argument types have to be in class *compare*.

To avoid the dependence on *compare*, we can also instruct *derive* to be based on *linorder*. Here, the command *derive linorder complex2* will create the instance *complex2 :: (linorder, linorder) linorder*, i.e., here the argument types have to be in class *linorder*.

```
derive compare-order complex
derive compare complex2
derive linorder complex2
derive countable complex complex2
derive equality complex
derive hashable complex complex2
```

**end**

## 10 Acknowledgements

We thank

- Lukas Bulwahn and Brian Huffman for the discussion on a generic derive command.
- Jasmin Blanchette for providing the tactic for countability for BNF-based datatypes.
- Jasmin Blanchette and Dmitriy Traytel for adjusting the Isabelle/ML interface of the BNF-based datatypes.
- Alexander Krauss for telling us to avoid the function package for this task.
- Peter Lammich for the inspiration of developing a hash-function generator.
- Andreas Lochbihler for the inspiration of developing generators for the container framework.
- Christian Urban for his cookbook on Isabelle/ML.

- Stefan Berghofer, Florian Haftmann, Cezary Kaliszyk, Tobias Nipkow, and Makarius Wenzel for their explanations on several Isabelle related questions.

## References

- [1] P. Lammich and A. Lochbihler. The Isabelle collections framework. In *Proc. ITP'10*, volume 6172 of *LNCS*, pages 339–354, 2010.
- [2] A. Lochbihler. Light-weight containers for isabelle: Efficient, extensible, nestable. In *Proc. ITP'13*, volume 7998 of *LNCS*, pages 116–132, 2013.
- [3] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.