

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` project¹ [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

1	Derive Manager	3
2	Shared Utilities for all Generator	3
3	Comparisons	4
3.1	Comparators and Linear Orders	4
3.2	Compare	6
3.3	Example: Modifying the Code-Equations of Red-Black-Trees	7
3.4	A Comparator-Interface to Red-Black-Trees	8

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

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

4	Generating Comparators	15
4.1	Lexicographic combination of <i>order</i>	15
4.2	Improved code for non-lazy languages	15
4.3	Pointwise properties for equality, symmetry, and transitivity .	15
4.4	Separate properties of comparators	17
4.5	Auxiliary Lemmas for Comparator Generator	19
4.6	The Comparator Generator	19
4.7	Compare Generator	19
4.8	Defining Comparators and Compare-Instances for Common Types	20
4.9	Defining Compare-Order-Instances for Common Types	21
4.10	Compare Instance for Rational Numbers	22
4.11	Compare Instance for Real Numbers	22
5	Checking Equality Without "="	22
5.1	Improved Code for Non-Lazy Languages	23
5.2	Partial Equality Property	23
5.3	Global equality property	23
5.4	The Generator	24
5.5	Defining Equality-Functions for Common Types	24
6	Generating Hash-Functions	25
6.1	Improved Code for Non-Lazy Languages	25
6.2	The Generator	25
6.3	Defining Hash-Functions for Common Types	25
7	Countable Datatypes	26
7.1	Installing the tactic	26
8	Loading Existing Derive-Commands	26
9	Examples	27
9.1	Rational Numbers	27
9.2	A Datatype Without Nested Recursion	27
9.3	Using Other datatypes	27
9.4	Mutual Recursion	28
9.5	Nested recursion	28
9.6	Examples from <i>lsaFoR</i>	28
9.7	A Complex Datatype	28
10	Acknowledgements	29

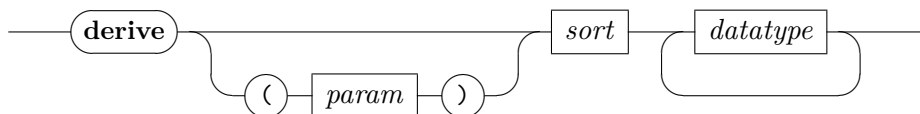
1 Derive Manager

```

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>

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>

lemma *in-set-simps*:

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

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

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

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

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

Ball (set (x # y # zs)) P = (P x ∧ Ball (set (y # zs)) P)
 ⟨proof⟩

lemma *conj-weak-cong*: a = b ⇒ c = d ⇒ (a ∧ c) = (b ∧ d) ⟨proof⟩

lemma *refl-True*: (x = x) = True ⟨proof⟩

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 *comparator* ⇒ 'a ⇒ 'a ⇒ *bool* **where**
lt-of-comp *acomp* x y = (case *acomp* x y of *Lt* ⇒ True | - ⇒ False)

definition *le-of-comp* :: 'a *comparator* ⇒ 'a ⇒ 'a ⇒ *bool* **where**
le-of-comp *acomp* x y = (case *acomp* x y of *Gt* ⇒ False | - ⇒ True)

definition *comp-of-ords* :: ('a ⇒ 'a ⇒ *bool*) ⇒ ('a ⇒ 'a ⇒ *bool*) ⇒ 'a *comparator* **where**
comp-of-ords *le* *lt* x y = (if *lt* x y then *Lt* else if *le* x y then *Eq* else *Gt*)

lemma *comp-of-ords-of-le-lt[simp]*: *comp-of-ords* (*le-of-comp* c) (*lt-of-comp* c) = c
 ⟨proof⟩

lemma *lt-of-comp-of-ords*: *lt-of-comp* (*comp-of-ords* *le* *lt*) = *lt*
 ⟨proof⟩

lemma *le-of-comp-of-ords-gen*: $(\bigwedge x y. \text{lt } x y \implies \text{le } x y) \implies \text{le-of-comp } (\text{comp-of-ords } \text{le } \text{lt}) = \text{le}$
 ⟨proof⟩

lemma *le-of-comp-of-ords-linorder*: **assumes** *class.linorder* *le* *lt*
shows *le-of-comp* (*comp-of-ords* *le* *lt*) = *le*
 ⟨proof⟩

```

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  $\implies$  x = y
    and comp-trans: comp x y = Lt  $\implies$  comp y z = Lt  $\implies$  comp x z = Lt
begin

lemma eq: (comp x y = Eq) = (x = y)
  <proof>

lemma comp-same [simp]:
  comp x x = Eq
  <proof>

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

sublocale ordering le lt
  <proof>

lemma linorder: class.linorder le lt
  <proof>

sublocale linorder le lt
  <proof>

lemma Gt-lt-conv: comp x y = Gt  $\longleftrightarrow$  lt y x
  <proof>
lemma Lt-lt-conv: comp x y = Lt  $\longleftrightarrow$  lt x y
  <proof>
lemma eq-Eq-conv: comp x y = Eq  $\longleftrightarrow$  x = y
  <proof>
lemma nGt-le-conv: comp x y  $\neq$  Gt  $\longleftrightarrow$  le x y
  <proof>
lemma nLt-le-conv: comp x y  $\neq$  Lt  $\longleftrightarrow$  le y x
  <proof>
lemma nEq-neq-conv: comp x y  $\neq$  Eq  $\longleftrightarrow$  x  $\neq$  y
  <proof>

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))
⟨proof⟩

```

end

```

lemma comp-of-ords: assumes class.linorder le lt
shows comparator (comp-of-ords le lt)
⟨proof⟩

```

```

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
⟨proof⟩

```

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

```

<proof>

lemma *compare-refl* [*simp*]:

compare x x = Eq

<proof>

end

lemma (**in** *linorder*) *le-lt-comparator-of*:

le-of-comp comparator-of = (\leq) lt-of-comp comparator-of = ($<$)

<proof>

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*

<proof>

context *compare-order*

begin

lemma *compare-is-comparator-of*:

compare = comparator-of

<proof>

lemmas *two-comparisons-into-compare* =

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

thm *two-comparisons-into-compare*

end

<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 ⇒ 'a → '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 ⇒ rbt-comp-lookup l k
  | Gt ⇒ rbt-comp-lookup r k
  | Eq ⇒ Some y)

fun

```


$rbt\text{-}comp\text{-}ins :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt$
where
 $rbt\text{-}comp\text{-}ins\ f\ k\ v\ RBT\text{-}Impl.\text{Empty} = Branch\ RBT\text{-}Impl.\text{R}\ RBT\text{-}Impl.\text{Empty}\ k\ v\ RBT\text{-}Impl.\text{Empty} \mid$
 $rbt\text{-}comp\text{-}ins\ f\ k\ v\ (Branch\ RBT\text{-}Impl.\text{B}\ l\ x\ y\ r) = (case\ c\ k\ x\ of$
 $\quad Lt \Rightarrow balance\ (rbt\text{-}comp\text{-}ins\ f\ k\ v\ l)\ x\ y\ r$
 $\quad \mid\ Gt \Rightarrow balance\ l\ x\ y\ (rbt\text{-}comp\text{-}ins\ f\ k\ v\ r)$
 $\quad \mid\ Eq \Rightarrow Branch\ RBT\text{-}Impl.\text{B}\ l\ x\ (f\ k\ y\ v)\ r) \mid$
 $rbt\text{-}comp\text{-}ins\ f\ k\ v\ (Branch\ RBT\text{-}Impl.\text{R}\ l\ x\ y\ r) = (case\ c\ k\ x\ of$
 $\quad Lt \Rightarrow Branch\ RBT\text{-}Impl.\text{R}\ (rbt\text{-}comp\text{-}ins\ f\ k\ v\ l)\ x\ y\ r$
 $\quad \mid\ Gt \Rightarrow Branch\ RBT\text{-}Impl.\text{R}\ l\ x\ y\ (rbt\text{-}comp\text{-}ins\ f\ k\ v\ r)$
 $\quad \mid\ Eq \Rightarrow Branch\ RBT\text{-}Impl.\text{R}\ l\ x\ (f\ k\ y\ v)\ r)$

definition $rbt\text{-}comp\text{-}insert\text{-}with\text{-}key :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt$
where $rbt\text{-}comp\text{-}insert\text{-}with\text{-}key\ f\ k\ v\ t = paint\ RBT\text{-}Impl.\text{B}\ (rbt\text{-}comp\text{-}ins\ f\ k\ v\ t)$

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

fun

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

where

$rbt\text{-}comp\text{-}del\ x\ RBT\text{-}Impl.\text{Empty} = RBT\text{-}Impl.\text{Empty} \mid$
 $rbt\text{-}comp\text{-}del\ x\ (Branch\ -\ a\ y\ s\ b) =$
 $\quad (case\ c\ x\ y\ of$
 $\quad\quad Lt \Rightarrow rbt\text{-}comp\text{-}del\text{-}from\text{-}left\ x\ a\ y\ s\ b$
 $\quad\quad \mid\ Gt \Rightarrow rbt\text{-}comp\text{-}del\text{-}from\text{-}right\ x\ a\ y\ s\ b$
 $\quad\quad \mid\ Eq \Rightarrow combine\ a\ b) \mid$
 $rbt\text{-}comp\text{-}del\text{-}from\text{-}left\ x\ (Branch\ RBT\text{-}Impl.\text{B}\ lt\ z\ v\ rt)\ y\ s\ b = balance\text{-}left$
 $(rbt\text{-}comp\text{-}del\ x\ (Branch\ RBT\text{-}Impl.\text{B}\ lt\ z\ v\ rt))\ y\ s\ b \mid$
 $rbt\text{-}comp\text{-}del\text{-}from\text{-}left\ x\ a\ y\ s\ b = Branch\ RBT\text{-}Impl.\text{R}\ (rbt\text{-}comp\text{-}del\ x\ a)\ y\ s\ b \mid$
 $rbt\text{-}comp\text{-}del\text{-}from\text{-}right\ x\ a\ y\ s\ (Branch\ RBT\text{-}Impl.\text{B}\ lt\ z\ v\ rt) = balance\text{-}right\ a$
 $y\ s\ (rbt\text{-}comp\text{-}del\ x\ (Branch\ RBT\text{-}Impl.\text{B}\ lt\ z\ v\ rt)) \mid$
 $rbt\text{-}comp\text{-}del\text{-}from\text{-}right\ x\ a\ y\ s\ b = Branch\ RBT\text{-}Impl.\text{R}\ a\ y\ s\ (rbt\text{-}comp\text{-}del\ x\ b)$

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

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

primrec

$rbt\text{-}comp\text{-}map\text{-}entry :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt$
where
 $rbt\text{-}comp\text{-}map\text{-}entry\ k\ f\ RBT\text{-}Impl.\text{Empty} = RBT\text{-}Impl.\text{Empty}$
 $\mid\ rbt\text{-}comp\text{-}map\text{-}entry\ k\ f\ (Branch\ cc\ lt\ x\ v\ rt) =$

```

(case c k x of
  Lt => Branch cc (rbt-comp-map-entry k f lt) x v rt
  | Gt => Branch cc lt x v (rbt-comp-map-entry k f rt)
  | Eq => Branch cc lt x (f v) rt)

```

function *comp-union-with* :: ('a => 'b => 'b => 'b) => ('a × 'b) list => ('a × 'b) list => ('a × 'b) list

where

```

comp-union-with f ((k, v) # as) ((k', v') # bs) =
  (case c k' k of
    Lt => (k', v') # comp-union-with f ((k, v) # as) bs
    | Gt => (k, v) # comp-union-with f as ((k', v') # bs)
    | Eq => (k, f k v v') # comp-union-with f as bs)
| comp-union-with f [] bs = bs
| comp-union-with f as [] = as
<proof>

```

termination <proof>

function *comp-sinter-with* :: ('a => 'b => 'b => 'b) => ('a × 'b) list => ('a × 'b) list => ('a × 'b) list

where

```

comp-sinter-with f ((k, v) # as) ((k', v') # bs) =
  (case c k' k of
    Lt => comp-sinter-with f ((k, v) # as) bs
    | Gt => comp-sinter-with f as ((k', v') # bs)
    | Eq => (k, f k v v') # comp-sinter-with f as bs)
| comp-sinter-with f [] - = []
| comp-sinter-with f - [] = []
<proof>

```

termination <proof>

fun *rbt-split-comp* :: ('a, 'b) rbt => 'a => ('a, 'b) rbt × 'b option × ('a, 'b) rbt

where

```

rbt-split-comp RBT-Impl.Empty k = (RBT-Impl.Empty, None, RBT-Impl.Empty)
| rbt-split-comp (RBT-Impl.Branch - l a b r) x = (case c x a of
  Lt => (case rbt-split-comp l x of (l1, β, l2) => (l1, β, rbt-join l2 a b r))
  | Gt => (case rbt-split-comp r x of (r1, β, r2) => (rbt-join l a b r1, β, r2))
  | Eq => (l, Some b, r))

```

lemma *rbt-split-comp-size*: (l2, b, r2) = rbt-split-comp t2 a => size l2 + size r2 ≤ size t2

<proof>

function *rbt-comp-union-rec* :: ('a => 'b => 'b => 'b) => ('a, 'b) rbt => ('a, 'b) rbt => ('a, 'b) rbt

where

```

rbt-comp-union-rec f t1 t2 = (let (f, t2, t1) =
  if flip-rbt t2 t1 then (λk v v'. f k v' v, t1, t2) else (f, t2, t1) in
  if small-rbt t2 then RBT-Impl.fold (rbt-comp-insert-with-key f) t2 t1
  else (case t1 of RBT-Impl.Empty => t2)

```

```

| RBT-Impl.Branch - l1 a b r1 ⇒
  case rbt-split-comp t2 a of (l2, β, r2) ⇒
    rbt-join (rbt-comp-union-rec f l1 l2) a (case β of None ⇒ b | Some b' ⇒ f
a b b') (rbt-comp-union-rec f r1 r2)))
⟨proof⟩
termination
⟨proof⟩

```

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

```

function rbt-comp-union-swap-rec :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ bool ⇒ ('a, 'b) rbt
⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt where
  rbt-comp-union-swap-rec f γ t1 t2 = (let (γ, t2, t1) =
    if flip-rbt t2 t1 then (¬γ, t1, t2) else (γ, t2, t1);
    f' = (if γ then (λk 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 ⇒ t2
  | Branch x l1 a b r1 ⇒
    case rbt-split-comp t2 a of (l2, β, r2) ⇒
      rbt-join (rbt-comp-union-swap-rec f γ l1 l2) a (case β of None ⇒ b | Some
x ⇒ f' a b x) (rbt-comp-union-swap-rec f γ r1 r2))
⟨proof⟩
termination
⟨proof⟩

```

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

```

lemma rbt-comp-union-swap-rec: rbt-comp-union-swap-rec f γ t1 t2 =
  rbt-comp-union-rec (if γ then (λk v v'. f k v' v) else f) t1 t2
⟨proof⟩

```

```

lemma rbt-comp-union-swap-rec-code[code]: rbt-comp-union-swap-rec f γ t1 t2 =
(
  let bh1 = bheight t1; bh2 = bheight t2; (γ, t2, bh2, t1, bh1) =
  if bh1 < bh2 then (¬γ, t1, bh1, t2, bh2) else (γ, t2, bh2, t1, bh1);
  f' = (if γ then (λk 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 ⇒ t2
  | Branch x l1 a b r1 ⇒
    case rbt-split-comp t2 a of (l2, β, r2) ⇒
      rbt-join (rbt-comp-union-swap-rec f γ l1 l2) a (case β of None ⇒ b | Some
x ⇒ f' a b x) (rbt-comp-union-swap-rec f γ r1 r2))
⟨proof⟩

```

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 (λ(k, v).
case rbt-comp-lookup t1 k of None ⇒ None

| *Some v' ⇒ Some (k, f k v' v)* (*RBT-Impl.entries t2*)

function *rbt-comp-inter-rec* :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt **where**
rbt-comp-inter-rec f t1 t2 = (let (f, t2, t1) =
 if *flip-rbt t2 t1* then (λk v v'. f k v' v, t1, t2) else (f, t2, t1) in
 if *small-rbt t2* then *rbtreeify (map-filter-comp-inter f t1 t2)*
 else case *t1* of *RBT-Impl.Empty* ⇒ *RBT-Impl.Empty*
 | *RBT-Impl.Branch* - *l1 a b r1* ⇒
 case *rbt-split-comp t2 a* of (*l2, β, r2*) ⇒ let *l' = rbt-comp-inter-rec f l1 l2; r' = rbt-comp-inter-rec f r1 r2* in
 (case *β* of *None* ⇒ *rbt-join2 l' r' | Some b' ⇒ rbt-join l' a (f a b b') r')*)
 ⟨*proof*⟩
termination
 ⟨*proof*⟩

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

function *rbt-comp-inter-swap-rec* :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ bool ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt **where**
rbt-comp-inter-swap-rec f γ t1 t2 = (let (γ, t2, t1) =
 if *flip-rbt t2 t1* then (¬γ, t1, t2) else (γ, t2, t1);
 f' = if γ then (λk v v'. f k v' v) else f in
 if *small-rbt t2* then *rbtreeify (map-filter-comp-inter f' t1 t2)*
 else case *t1* of *rbt.Empty* ⇒ *rbt.Empty*
 | *Branch x l1 a b r1* ⇒
 (case *rbt-split-comp t2 a* of (*l2, β, r2*) ⇒ let *l' = rbt-comp-inter-swap-rec f γ l1 l2; r' = rbt-comp-inter-swap-rec f γ r1 r2* in
 (case *β* of *None* ⇒ *rbt-join2 l' r' | Some b' ⇒ rbt-join l' a (f' a b b') r'))*)
 ⟨*proof*⟩
termination
 ⟨*proof*⟩

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

lemma *rbt-comp-inter-swap-rec: rbt-comp-inter-swap-rec f γ t1 t2 = rbt-comp-inter-rec (if γ then (λk v v'. f k v' v) else f) t1 t2*
 ⟨*proof*⟩

lemma *comp-inter-with-key-code*[*code*]: *rbt-comp-inter-swap-rec f γ t1 t2 = (let bh1 = bheight t1; bh2 = bheight t2; (γ, t2, bh2, t1, bh1) = if bh1 < bh2 then (¬γ, t1, bh1, t2, bh2) else (γ, t2, bh2, t1, bh1); f' = (if γ then (λk v v'. f k v' v) else f) in if bh2 < 4 then rbtreeify (map-filter-comp-inter f' t1 t2) else case t1 of rbt.Empty ⇒ rbt.Empty | Branch x l1 a b r1 ⇒ (case rbt-split-comp t2 a of (l2, β, r2) ⇒ let l' = rbt-comp-inter-swap-rec f γ l1 l2; r' = rbt-comp-inter-swap-rec f γ r1 r2 in (case β of None ⇒ rbt-join2 l' r' | Some b' ⇒ rbt-join l' a (f' a b b') r'))*)

<proof>

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

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

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

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

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

context

assumes c : *comparator* c

begin

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

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

lemma *rbt-comp-insert-with-key*: $\text{rbt-comp-insert-with-key} = \text{ord.rbt-insert-with-key } (\text{lt-of-comp } c)$
<proof>

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

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

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

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

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

<proof>

lemma *comp-sunion-with*: *comp-sunion-with* = *ord.sunion-with* (*lt-of-comp* *c*)
<proof>

lemma *anti-sym*: *lt-of-comp* *c* *a* *x* \implies *lt-of-comp* *c* *x* *a* \implies *False*
<proof>

lemma *rbt-split-comp*: *rbt-split-comp* *t* *x* = *ord.rbt-split* (*lt-of-comp* *c*) *t* *x*
<proof>

lemma *comp-union-with-key*: *rbt-comp-union-rec* *f* *t1* *t2* = *ord.rbt-union-rec* (*lt-of-comp* *c*) *f* *t1* *t2*
<proof>

lemma *comp-sinter-with*: *comp-sinter-with* = *ord.sinter-with* (*lt-of-comp* *c*)
<proof>

lemma *rbt-comp-union-with-key*: *rbt-comp-union-with-key* = *ord.rbt-union-with-key* (*lt-of-comp* *c*)
<proof>

lemma *comp-inter-with-key*: *rbt-comp-inter-rec* *f* *t1* *t2* = *ord.rbt-inter-rec* (*lt-of-comp* *c*) *f* *t1* *t2*
<proof>

lemma *rbt-comp-inter-with-key*: *rbt-comp-inter-with-key* = *ord.rbt-inter-with-key* (*lt-of-comp* *c*)
<proof>

lemma *comp-minus*: *comp-minus* *t1* *t2* = *ord.rbt-minus-rec* (*lt-of-comp* *c*) *t1* *t2*
<proof>

lemma *rbt-comp-minus*: *rbt-comp-minus* = *ord.rbt-minus* (*lt-of-comp* *c*)
<proof>

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 \Rightarrow 'b comparator \Rightarrow 'c comparator \Rightarrow 'z comparator \Rightarrow ('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)
  <proof>

```

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)
  <proof>

```

```

definition trans-order :: order  $\Rightarrow$  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 $x\ y\ z$
<proof>

lemma *trans-orderD*:

assumes *trans-order* $x\ y\ z$ **and** $x \neq Gt$ **and** $y \neq Gt$
shows $z \neq Gt$ **and** $x = Lt \vee y = Lt \implies z = Lt$
<proof>

lemma *All-less-Suc*:

$(\forall i < Suc\ x. P\ i) \longleftrightarrow P\ 0 \wedge (\forall i < x. P\ (Suc\ i))$
<proof>

lemma *comp-lex-trans*:

assumes *length* $xs = length\ ys$
and *length* $ys = length\ zs$
and $\forall i < length\ zs. trans_order\ (xs\ !\ i)\ (ys\ !\ i)\ (zs\ !\ i)$
shows *trans-order* $(comp_lex\ xs)\ (comp_lex\ ys)\ (comp_lex\ zs)$
<proof>

lemma *comp-lex-sym*:

assumes *length* $xs = length\ ys$
and $\forall i < length\ ys. invert_order\ (xs\ !\ i) = ys\ !\ i$
shows *invert-order* $(comp_lex\ xs) = comp_lex\ ys$
<proof>

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

definition *peq-comp* :: $'a\ comparator \Rightarrow 'a \Rightarrow bool$

where

peq-comp *acomp* $x \longleftrightarrow (\forall y. acomp\ x\ y = Eq \longleftrightarrow x = y)$

lemma *peq-compD*: *peq-comp* *acomp* $x \implies acomp\ x\ y = Eq \longleftrightarrow x = y$

<proof>

lemma *peq-compI*: $(\bigwedge y. acomp\ x\ y = Eq \longleftrightarrow x = y) \implies peq_comp\ acomp\ x$

<proof>

definition *psym-comp* :: $'a\ comparator \Rightarrow 'a \Rightarrow bool$ **where**

psym-comp *acomp* $x \longleftrightarrow (\forall y. invert_order\ (acomp\ x\ y) = (acomp\ y\ x))$

lemma *psym-compD*:

assumes *psym-comp* *acomp* x
shows *invert-order* $(acomp\ x\ y) = (acomp\ y\ x)$
<proof>

lemma *psym-compI*:

assumes $\bigwedge y. \text{invert-order } (\text{acom}p\ x\ y) = (\text{acom}p\ y\ x)$
shows $\text{psym-comp } \text{acom}p\ x$
 $\langle \text{proof} \rangle$

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

lemma ptrans-compD :
assumes $\text{ptrans-comp } \text{acom}p\ x$
shows $\text{trans-order } (\text{acom}p\ x\ y) (\text{acom}p\ y\ z) (\text{acom}p\ x\ z)$
 $\langle \text{proof} \rangle$

lemma ptrans-compI :
assumes $\bigwedge y\ z. \text{trans-order } (\text{acom}p\ x\ y) (\text{acom}p\ y\ z) (\text{acom}p\ x\ z)$
shows $\text{ptrans-comp } \text{acom}p\ x$
 $\langle \text{proof} \rangle$

4.4 Separate properties of comparators

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

lemma eq-compD2 : $\text{eq-comp } \text{acom}p \Longrightarrow \text{peq-comp } \text{acom}p\ x$
 $\langle \text{proof} \rangle$

lemma eq-compI2 : $(\bigwedge x. \text{peq-comp } \text{acom}p\ x) \Longrightarrow \text{eq-comp } \text{acom}p$
 $\langle \text{proof} \rangle$

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

lemma trans-compD2 : $\text{trans-comp } \text{acom}p \Longrightarrow \text{ptrans-comp } \text{acom}p\ x$
 $\langle \text{proof} \rangle$

lemma trans-compI2 : $(\bigwedge x. \text{ptrans-comp } \text{acom}p\ x) \Longrightarrow \text{trans-comp } \text{acom}p$
 $\langle \text{proof} \rangle$

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

lemma sym-compD2 :
 $\text{sym-comp } \text{acom}p \Longrightarrow \text{psym-comp } \text{acom}p\ x$
 $\langle \text{proof} \rangle$

lemma sym-compI2 : $(\bigwedge x. \text{psym-comp } \text{acom}p\ x) \Longrightarrow \text{sym-comp } \text{acom}p$
 $\langle \text{proof} \rangle$

lemma *eq-compD*: $eq\text{-}comp\ acomp \implies acomp\ x\ y = Eq \longleftrightarrow x = y$
 ⟨*proof*⟩

lemma *eq-compI*: $(\bigwedge x\ y. acomp\ x\ y = Eq \longleftrightarrow x = y) \implies eq\text{-}comp\ acomp$
 ⟨*proof*⟩

lemma *trans-compD*: $trans\text{-}comp\ acomp \implies trans\text{-}order\ (acom\ x\ y)\ (acom\ y\ z)$
 ($acom\ x\ z$)
 ⟨*proof*⟩

lemma *trans-compI*: $(\bigwedge x\ y\ z. trans\text{-}order\ (acom\ x\ y)\ (acom\ y\ z)\ (acom\ x\ z))$
 $\implies trans\text{-}comp\ acomp$
 ⟨*proof*⟩

lemma *sym-compD*:
 $sym\text{-}comp\ acomp \implies invert\text{-}order\ (acom\ x\ y) = (acom\ y\ x)$
 ⟨*proof*⟩

lemma *sym-compI*: $(\bigwedge x\ y. invert\text{-}order\ (acom\ x\ y) = (acom\ y\ x)) \implies sym\text{-}comp\ acomp$
 ⟨*proof*⟩

lemma *eq-sym-trans-imp-comparator*:
assumes *eq-comp acomp and sym-comp acomp and trans-comp acomp*
shows *comparator acomp*
 ⟨*proof*⟩

lemma *comparator-imp-eq-sym-trans*:
assumes *comparator acomp*
shows *eq-comp acomp sym-comp acomp trans-comp acomp*
 ⟨*proof*⟩

context
fixes *acomp :: 'a comparator*
assumes *c: comparator acomp*
begin

lemma *comp-to-psym-comp*: $psym\text{-}comp\ acomp\ x$
 ⟨*proof*⟩

lemma *comp-to-peq-comp*: $peq\text{-}comp\ acomp\ x$
 ⟨*proof*⟩

lemma *comp-to-ptrans-comp*: $ptrans\text{-}comp\ acomp\ x$
 ⟨*proof*⟩

end

4.5 Auxiliary Lemmas for Comparator Generator

lemma *forall-finite*: $(\forall i < (0 :: nat). P i) = True$
 $(\forall i < Suc 0. P i) = P 0$
 $(\forall i < Suc (Suc x). P i) = (P 0 \wedge (\forall i < Suc x. P (Suc i)))$
{proof}

lemma *trans-order-different*:
trans-order a b Lt
trans-order Gt b c
trans-order a Gt c
{proof}

lemma *length-nth-simps*:
 $length [] = 0$ $length (x \# xs) = Suc (length xs)$
 $(x \# xs) ! 0 = x$ $(x \# xs) ! (Suc n) = xs ! n$ {proof}

4.6 The 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

$lt\ x\ y = (le\ x\ y \wedge \neg le\ y\ x)$ (**is** *?a*)

$le\ x\ x$ (**is** *?b*)

$le\ x\ y \implies le\ y\ z \implies le\ x\ z$ (**is** *?c1 \implies ?c2 \implies ?c3*)

$le\ x\ y \implies le\ y\ x \implies x = y$ (**is** *?d1 \implies ?d2 \implies ?d3*)

$le\ x\ y \vee le\ y\ x$ (**is** *?e*)

<proof>

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

<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
  <proof>

```

```

lemma comparator-bool: comparator comparator-bool
  <proof>

```

<ML>

```

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>

```

```

lemma comparator-unit-comparator-of [compare-simps]:
  comparator-unit = comparator-of
  <proof>

```

```

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⟩
```

```
instance prod :: (compare-order, compare-order)compare-order
⟨proof⟩
```

```
instance option :: (compare-order)compare-order
⟨proof⟩
```

```
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  
⟨proof⟩
```

```
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 $(\text{'a}, \text{'b}, \text{'c}, \text{'z})$ *Equality-Generator.type* constructs an equality-test function of type $(\text{'a} \Rightarrow \text{'a} \Rightarrow \text{bool}) \Rightarrow (\text{'b} \Rightarrow \text{'b} \Rightarrow \text{bool}) \Rightarrow (\text{'c} \Rightarrow \text{'c} \Rightarrow \text{bool}) \Rightarrow (\text{'z} \Rightarrow \text{'z} \Rightarrow \text{bool}) \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'z})$ *Equality-Generator.type* $\Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'z})$ *Equality-Generator.type* $\Rightarrow \text{bool}$. 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*:

$$\text{list-all-eq } [] = \text{True}$$

$$\text{list-all-eq } [b] = b$$

$$\text{list-all-eq } (b1 \# b2 \# bs) = (b1 \wedge \text{list-all-eq } (b2 \# bs))$$

<proof>

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

<proof>

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

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

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

<proof>

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

<proof>

5.3 Global equality property

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

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

lemma *equalityD2*: $\text{equality aeq} \implies \text{pequality aeq } x$
⟨*proof*⟩

lemma *equalityI2*: $(\bigwedge x. \text{pequality aeq } x) \implies \text{equality aeq}$
⟨*proof*⟩

lemma *equalityD*: $\text{equality aeq} \implies \text{aeq } x \ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *equalityI*: $(\bigwedge x \ y. \text{aeq } x \ y \longleftrightarrow x = y) \implies \text{equality aeq}$
⟨*proof*⟩

lemma *equality-imp-eq*:
 $\text{equality aeq} \implies \text{aeq } = (=)$
⟨*proof*⟩

lemma *eq-equality*: $\text{equality } (=)$
⟨*proof*⟩

lemma *equality-def'*: $\text{equality } f = (f = (=))$
⟨*proof*⟩

5.4 The 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
  <proof>
```

6.2 The 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.

<ML>

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
```

```
  (proof)
```

```
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 = MEEmpty | 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 = TEEmpty | 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.