Deriving class instances for datatypes.*

Christian Sternagel and René Thiemann

February 23, 2021

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\(^1\) [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 ................................... 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.
\(^1\)http://cl-informatik.uibk.ac.at/software/ceta
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:
  \( x \in \text{set} \ (y \# z \# ys) = (x = y \lor x \in \text{set} \ (z \# ys)) \)
  \( x \in \text{set} \ ([y]) = (x = y) \)
  \( x \in \text{set} \ [] = \text{False} \)
  \( \text{Ball} \ (\text{set} \ []) \ P = \text{True} \)
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 ⇒ c = d ⇒ (a ∧ c) = (b ∧ d) by auto

lemma refl-True: (x = x) = 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, ((<), (≤)),
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 be-	ween 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
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-of-comp (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: (∀ x y. lt x y ⇒ le x y) ⇒ le-of-comp (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 ⇒ 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 ⇒ x = y
and trans: comp x y = Lt ⇒ comp y z = Lt ⇒ 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 ≡ lt-of-comp comp
abbreviation le ≡ le-of-comp comp

lemma linorder: class.linorder le lt
proof
note [simp] = lt-of-comp-def le-of-comp-def
fix x y z :: 'a
show lt x y = (le x y ∧ ¬ le y x)
  using sym[of x y] by (cases comp x y) (simp-all)
show le x y ∨ le y x
  using sym[of x y] by (cases comp x y) (simp-all)
show le x x using eq[of x x] by (simp)
show le x y ⇒ le y x ⇒ x = y
  using sym[of x y] by (cases comp x y) (simp-all add: eq)
show le x y ⇒ le y z ⇒ le x z
  by (cases comp x y comp y z rule: order.exhaust [case-product order.exhaust])
(auto dest: trans simp: eq)
qed
sublocale linorder le lt  
  by (rule linorder)

lemma Gt-lt-conv: comp x y = Gt ←→ 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 ←→ lt x y  
  unfolding lt-of-comp-def by (cases comp x y, auto)

lemma eq-Eq-conv: comp x y = Eq ←→ x = y  
  by (rule eq)

lemma nGt-le-conv: comp x y ≠ Gt ←→ le x y  
  unfolding le-of-comp-def by (cases comp x y, auto)

lemma nLt-le-conv: comp x y ≠ Lt ←→ 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 ≠ Eq ←→ x ≠ y  
  using eq-Eq-conv[of x y] by simp

lemmas le-lt-convs = 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 R) else P) = (case-order P Q R (comp x y))  
  (if lt x y then Q else (if le y x then R else Q)) = (case-order P Q R (comp x y))  
  (if lt x y then Q else (if lt x y then R else Q)) = (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 le y x then Q else R)) = (case-order P Q R (comp x y))  
  by (auto simp: le-lt-convs 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
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 ↔ 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 = (≤) 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 = (≤) 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:

end
\textit{compare} = \textit{comparator-of}

\textbf{proof} (\textit{intro ext})

\textbf{fix} \( x \ y :: \ 'a \)

\textbf{show} \( \text{compare} \ x \ y = \text{comparator-of} \ x \ y \)

\textbf{by} (\textit{unfold} \textit{comparator-of-def}, \textit{unfold} \textit{ord-defs}[\textit{symmetric}] \textit{lt-of-comp-def},

\textit{cases} \textit{compare} \ x \ y, \textit{auto})

\textbf{qed}

\textbf{lemmas} \textit{two-comparisons-into-compare} =

\textit{comparator-two-comparisons-into-case-order}[\textit{OF} \textit{comparator-compare}, \textit{unfoldord-defs}]

\textbf{thm} \textit{two-comparisons-into-compare}

\textbf{end}

\textbf{ML-file} (\textit{compare-code.ML})

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

\textbf{end}

\textbf{3.3 Example: Modifying the Code-Equations of Red-Black-Trees}

\textbf{theory} \textit{RBT-Compare-Order-Impl}

\textbf{imports}

\textit{Compare}

\textit{HOL-Library.RBT-Impl}

\textbf{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 \textit{comparator}, where before two comparisons have been performed. The disadvantage of this simple solution is the additional class constraint on \textit{compare-order}.

\textit{compare-code} \((a)\ rbt-ins\)

\textit{compare-code} \((a)\ rbt-lookup\)

\textit{compare-code} \((a)\ rbt-del\)

\textit{compare-code} \((a)\ rbt-map-entry\)

\textit{compare-code} \((a)\ sunion-with\)

\textit{compare-code} \((a)\ sinters-with\)

\textit{export-code} \textit{rbt-ins\ rbt-lookup\ rbt-del\ rbt-map-entry\ rbt-union-with-key\ rbt-inter-with-key\ in\ Haskell}
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.

case 
  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-comp-ins :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
where 
  rbt-comp-ins f k v t = paint RBT-Impl.B l x y r |
  rbt-comp-ins f k v (Branch RBT-Impl.B l x y r) = (case c k x of
    Lt ⇒ balance (rbt-comp-ins f k v l) x y r |
    Gt ⇒ balance l x y (rbt-comp-ins f k v r) |
    Eq ⇒ Branch RBT-Impl.B l x y (rbt-comp-ins f k v r))

definition rbt-comp-insert-with-key :: ('a ⇒ 'b ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
where 
  rbt-comp-insert-with-key f k v t = paint RBT-Impl.B (rbt-comp-ins f k v t)

definition rbt-comp-insert :: 'a ⇒ 'b ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
where 
  rbt-comp-insert = rbt-comp-insert-with-key (λ- - nv. nv)
fun
rbt-comp-del-from-left :: 'a ⇒ ('a,'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt
and
rbt-comp-del-from-right :: 'a ⇒ ('a,'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt
and
rbt-comp-del :: 'a⇒ ('a,'b) rbt ⇒ ('a,'b) rbt
where
rbt-comp-del x RBT-Impl.Empty = RBT-Impl.Empty |
rbt-comp-del x (Branch - a y s b) =
  (case c y of
     Lt ⇒ rbt-comp-del-from-left x a y s b |
     Gt ⇒ rbt-comp-del-from-right x a y s b |
     Eq ⇒ combine a b) |
rbt-comp-del-from-left x (Branch RBT-Impl.B lt z v rt) y s b = balance-left
(rbtk-comp-del x (Branch RBT-Impl.B lt z v rt)) y s b |
rbt-comp-del-from-left x a y s b = Branch RBT-Impl.R (rbt-comp-del x a) y s b |
rbt-comp-del-from-right x a y s (Branch RBT-Impl.B lt z v rt) = balance-right a y s (rbt-comp-del x (Branch RBT-Impl.B lt z v rt)) |
rbt-comp-del-from-right x a y s b = Branch RBT-Impl.R a y s (rbt-comp-del x b)
definition rbt-comp-delete k t = paint RBT-Impl.B (rbt-comp-del k t)
definition rbt-comp-bulkload xs = foldr (λ(k, v). rbt-comp-insert k v) xs RBT-Impl.Empty
primrec
rbt-comp-map-entry :: 'a ⇒ ('b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
where
rbt-comp-map-entry k f RBT-Impl.Empty = RBT-Impl.Empty |
rbt-comp-map-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)
definition comp-sunion-with :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a × 'b) list
list ⇒ ('a × 'b) list
where
comp-sunion-with f ((k, v) # as) ((k', v') # bs) =
  (case c k' k of
     Lt ⇒ (k', v') # comp-sunion-with f ((k, v) # as) bs |
     Gt ⇒ (k, v) # comp-sunion-with f as ((k', v') # bs) |
     Eq ⇒ (k, f k v v') # comp-sunion-with f as bs) |
comp-sunion-with f [] bs = bs |
comp-sunion-with f as [] = as
by pat-completeness auto
termination by lexicographic-order
function comp-sinter-with :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a × 'b) list
\[
\Rightarrow ('a \times 'b) \text{ list}
\]

**where**

\[
\text{comp-sinter-with } f ((k, v) \# as) ((k', v') \# bs) =
\]

(case c k' k of
  \[
  \begin{array}{ll}
  Lt & \Rightarrow \text{comp-sinter-with } f ((k, v) \# as) bs \\
  Gt & \Rightarrow \text{comp-sinter-with } f as ((k', v') \# bs) \\
  Eq & \Rightarrow (k, f k v v') \# \text{comp-sinter-with } f as bs
  \end{array}
\]

by \text{pat-completeness auto}

**termination by** lexicographic-order

**definition** \text{rbt-comp-union-with-key} :: ('a => 'b => 'b => 'b) => ('a, 'b) rbt => ('a, 'b) rbt

**where**

\[
\text{rbt-comp-union-with-key } f \ t1 \ t2 =
\]

(case \text{RBT-Impl.compare-height } t1 \ t1 \ t2 \ t2
  of \text{compare.EQ} \Rightarrow \text{rbt-simpl} ((\text{comp-sunion-with } f (\text{RBT-Impl.entries } t1) (\text{RBT-Impl.entries } t2))
  \mid \text{compare.LT} \Rightarrow \text{RBT-Impl.fold} (\text{rbt-comp-insert-with-key} (\lambda k v w. f k v w)) t1
t2
  \mid \text{compare.GT} \Rightarrow \text{RBT-Impl.fold} (\text{rbt-comp-insert-with-key } f) t2 t1
\]

**definition** \text{rbt-comp-inter-with-key} :: ('a => 'b => 'b => 'b) => ('a, 'b) rbt => ('a, 'b) rbt

**where**

\[
\text{rbt-comp-inter-with-key } f \ t1 \ t2 =
\]

(case \text{RBT-Impl.compare-height } t1 \ t1 \ t2 \ t2
  of \text{compare.EQ} \Rightarrow \text{rbt-simpl} ((\text{comp-sinter-with } f (\text{RBT-Impl.entries } t1) (\text{RBT-Impl.entries } t2))
  \mid \text{compare.LT} \Rightarrow \text{rbt-simpl} (\text{List.map-filter} (\lambda(k, v). \text{map-option} (\lambda w. (k, f k v w)) (\text{rbt-comp-lookup } t2 k)) (\text{RBT-Impl.entries } t1))
  \mid \text{compare.GT} \Rightarrow \text{rbt-simpl} (\text{List.map-filter} (\lambda(k, v). \text{map-option} (\lambda w. (k, f k w v)) (\text{rbt-comp-lookup } t1 k)) (\text{RBT-Impl.entries } t2))
\]

**context**

assumes \text{c} : \text{comparator c}

**begin**

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

**proof** (intro ext)

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

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

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

  (lt-of-comp c)
unfolding rbt-comp-insert-with-key-def[abs-def] ord.rbt-insert-with-key-def[abs-def]
unfolding rbt-comp-ins ..

lemma rbt-comp-insert: rbt-comp-insert = ord.rbt-insert (lt-of-comp c)
unfolding rbt-comp-insert-with-key ..

lemma rbt-comp-del: rbt-comp-del = ord.rbt-del (lt-of-comp c)
proof −
  { fix k a b and s t :: ('a,'b)rbt
    have
      rbt-comp-del-from-left k t a b s = ord.rbt-del-from-left (lt-of-comp c) k t a b s
      rbt-comp-del-from-right k t a b s = ord.rbt-del-from-right (lt-of-comp c) k t a b s
      rbt-comp-del k t = ord.rbt-del (lt-of-comp c) k t
    by (induct k t a b s and k t a b s and k t rule: rbt-comp-del-from-left-rbt-comp-del-from-right-rbt-comp-del.induct
      unfold
      rbt-comp-del.simps ord.rbt-del.simps
      rbt-comp-del-from-left.simps ord.rbt-del-from-left.simps
      rbt-comp-del-from-right.simps ord.rbt-del-from-right.simps
      comparator:two-comparisons-into-case-order[OF c],
      auto split: order.split)
  }
  thus ?thesis by (intro ext)
qed

lemma rbt-comp-delete: rbt-comp-delete = ord.rbt-delete (lt-of-comp c)
unfolding rbt-comp-delete-def[abs-def] ord.rbt-delete-def[abs-def]
unfolding rbt-comp-del ..

lemma rbt-comp-bulkload: rbt-comp-bulkload = ord.rbt-bulkload (lt-of-comp c)
unfolding rbt-comp-bulkload-def[abs-def] ord.rbt-bulkload-def[abs-def]
unfolding rbt-comp-insert ..

lemma rbt-comp-map-entry: rbt-comp-map-entry = ord.rbt-map-entry (lt-of-comp c)
proof (intro ext)
  fix f k and t :: ('a,'b)rbt
  show rbt-comp-map-entry f k t = ord.rbt-map-entry (lt-of-comp c) f k t
by (induct t, unfold rbt-comp-map-entry.simps ord.rbt-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 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)
unfolding rbt-comp-union-with-key-def[abs-def] ord.rbt-union-with-key-def[abs-def]
unfolding rbt-comp-insert-with-key comp-sunion-with ..

lemma rbt-comp-inter-with-key: rbt-comp-inter-with-key = ord.rbt-inter-with-key
(lt-of-comp c)
unfolding rbt-comp-inter-with-key-def[abs-def] ord.rbt-inter-with-key-def[abs-def]
unfolding rbt-comp-insert-with-key comp-sinter-with rbt-comp-lookup ..

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
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 ⇒ order
where
comp-lex (c # cs) = (case c of Eq ⇒ comp-lex cs | - ⇒ 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 ⇒ comp-lex (d # cs) | z ⇒ z)
by (cases c, auto)+

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 <-> (∀ ord ∈ set os. ord = Eq)
by (induct os) (auto split: order.splits)

definition trans-order :: order ⇒ order ⇒ order ⇒ bool where
trans-order x y z <-> x ≠ Gt ⇒ y ≠ Gt ⇒ z ≠ Gt ∧ ((x = Lt ∨ y = Lt) ⇒ z = Lt)
lemma trans-orderI:
  \((x \neq Gt \implies y \neq Gt \implies z \neq Gt \land ((x = Lt \lor y = Lt) \implies z = Lt))) \implies\ntrans-order\ x\ y\ z\\nby\ (simp\ add:\ trans-order-def)\n
lemma trans-orderD:
  assumes \(trans-order\ x\ y\ z\) and \(x \neq Gt\) and \(y \neq Gt\)
  shows \(z \neq Gt\) and \(x = Lt \lor y = Lt \implies z = Lt\)
  using assms by (auto simp: trans-order-def)

lemma All-less-Suc:
  \((\forall i < Suc\ x.\ P\ i) \iff P\ 0 \land (\forall i < x.\ P\ (Suc\ i))\)
  using less-Suc-eq-0-disj by force

lemma comp-lex-trans:
  assumes \(length\ xs = length\ ys\) and \(\forall i < length\ zs.\ trans-order\ (xs\ !\ i)\ (ys\ !\ i)\ (zs\ !\ i)\)
  shows \(trans-order\ (comp-lex\ xs)\ (comp-lex\ ys)\ (comp-lex\ zs)\)
  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 \(\forall i < length\ ys.\ invert-order\ (xs\ !\ i) = (ys\ !\ i)\)
  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 p eq-comp :: 'a comparator \Rightarrow 'a \Rightarrow bool
where
  p eq-comp acomp x \iff (\forall y. acomp x y = Eq \iff x = y)

lemma p eq-compD: p eq-comp acomp x \implies acomp x y = Eq \iff x = y
unfolding p eq-comp-def by auto

lemma p eq-compI: (\land y. acomp x y = Eq \iff x = y) \implies p eq-comp acomp x
unfolding p eq-comp-def by auto

definition p sym-comp :: 'a comparator \Rightarrow 'a \Rightarrow bool
where
  p sym-comp acomp x \iff (\forall y. invert-order (acomp x y) = (acomp y x))

15
lemma \texttt{psym-compD}:
assumes \texttt{psym-comp acomp x}
shows invert-order (acomp x y) = (acomp y x)
using \texttt{assms unfolding psym-comp-def by blast}

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

\begin{definition}
\texttt{ptrans-comp} :: \texttt{'a comparator \Rightarrow 'a \\Rightarrow bool}
\end{definition}
\texttt{ptrans-comp acomp x} \iff \(
\forall y z. \text{trans-order} (acomp x y) (acomp y z) (acomp x z)
\)

lemma \texttt{ptrans-compD}:
assumes \texttt{ptrans-comp acomp x}
shows trans-order (acomp x y) (acomp y z) (acomp x z)
using \texttt{assms unfolding ptrans-comp-def by blast}

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

4.4 Separate properties of comparators

\begin{definition}
\texttt{eq-comp} :: \texttt{'a comparator \Rightarrow bool}
\end{definition}
eq-comp acomp \iff \(\forall x. \text{peq-comp acomp x}\)

lemma \texttt{eq-compD2}: eq-comp acomp \implies peq-comp acomp x
unfolding eq-comp-def by blast

lemma \texttt{eq-compI2}: (\forall x. \text{peq-comp acomp x}) \implies eq-comp acomp
unfolding eq-comp-def by blast

\begin{definition}
\texttt{trans-comp} :: \texttt{'a comparator \Rightarrow bool}
\end{definition}
\texttt{trans-comp acomp} \iff \(\forall x. \text{ptrans-comp acomp x}\)

lemma \texttt{trans-compD2}: trans-comp acomp \implies ptrans-comp acomp x
unfolding trans-comp-def by blast

lemma \texttt{trans-compI2}: (\forall x. \text{ptrans-comp acomp x}) \implies trans-comp acomp
unfolding trans-comp-def by blast

\begin{definition}
\texttt{sym-comp} :: \texttt{'a comparator \Rightarrow bool}
\end{definition}
sym-comp acomp \iff \(\forall x. \text{psym-comp acomp x}\)
lemma sym-compD2:
  \( \text{sym-comp } a\text{comp } \implies \text{psym-comp } a\text{comp } x \)
  unfolding sym-comp-def by blast

lemma sym-compI2: \((\forall x. \text{psym-comp } a\text{comp } x) \implies \text{sym-comp } a\text{comp }\)
  unfolding sym-comp-def by blast

lemma eq-compD: \(\text{eq-comp } a\text{comp } \implies a\text{comp } x y = \text{Eq } \iff x = y\)
  by (rule peq-compD[OF eq-compD2])

lemma eq-compI: \((\forall x y. a\text{comp } x y = \text{Eq } \iff x = y) \implies \text{eq-comp } a\text{comp }\)
  by (intro eq-compI2 peq-compI)

lemma trans-compD: \(\text{trans-comp } a\text{comp } \implies \text{trans-order } (a\text{comp } x y) (a\text{comp } y z) (a\text{comp } x z)\)
  by (rule ptrans-compD[OF trans-compD2])

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

lemma sym-compD:
  \(\text{sym-comp } a\text{comp } \implies \text{invert-order } (a\text{comp } x y) = (a\text{comp } y x)\)
  by (rule psym-compD[OF sym-compD2])

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

lemma eq-sym-trans-imp-comparator:
  assumes eq-comp acomp and sym-comp acomp and trans-comp acomp
  shows comparator acomp
proof
  fix x y z
  show invert-order (a\text{comp } x y) = a\text{comp } y x
    using sym-compD [OF \(\text{sym-comp } a\text{comp}\)] .
    \{
      assume a\text{comp } x y = \text{Eq }
      with eq-compD [OF eq-compD2]
      show x = y by blast
    \}
    \{
      assume a\text{comp } x y = \text{Lt } and a\text{comp } y z = \text{Lt }
      with trans-orderD [OF trans-compD2] [OF \(\text{trans-comp } a\text{comp}\)], of x y z
      show a\text{comp } x z = \text{Lt } by auto
    \}
qed
lemma comparator-imp-eq-sym-trans:
assumes comparator acomp
shows eq-comp acomp sym-comp acomp trans-comp acomp
proof −
interpret comparator acomp by fact
show eq-comp acomp using eq by (intro eq-compI, auto)
show sym-comp acomp using sym by (intro sym-compI, auto)
show trans-comp acomp
proof (intro trans-compI trans-orderI)
fix x y z
assume acomp x y ≠ Gt acomp y z ≠ Gt
thus acomp x z ≠ Gt ∧ (acomp x y = Lt ∨ acomp y z = Lt ---→ acomp x z = Lt)
using trans [of x y z] and eq [of x y] and eq [of y z]
by (cases acomp x y acomp 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:
\[
\begin{align*}
\text{length} [] &= 0 \\
\text{length} (x # xs) &= \text{Suc} \ (\text{length} \ xs) \\
(x # xs)! 0 &= x \\
(x # xs)! (\text{Suc} \ n) &= xs! n \quad \text{by auto}
\end{align*}
\]

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
\[
\begin{align*}
\text{lt} \ x \ y &= (\text{le} \ x \ y \ \land \ \lnot \ \text{le} \ y \ x) \quad (\text{is } \ ?a) \\
\text{le} \ x \ x &= (\text{is } \ ?b) \\
\text{le} \ x \ y \ \Longrightarrow \ \text{le} \ y \ z \ \Longrightarrow \ \text{le} \ x \ z &= (\text{is } \ ?c1 \ \Longrightarrow \ ?c2 \ \Longrightarrow \ ?c3) \\
\text{le} \ x \ y \ \Longrightarrow \ \text{le} \ y \ x \ \Longrightarrow \ x = y &= (\text{is } \ ?d1 \ \Longrightarrow \ ?d2 \ \Longrightarrow \ ?d3) \\
\text{le} \ x \ y \ \lor \ \text{le} \ y \ x &= (\text{is } \ ?e)
\end{align*}
\]
proof 
  interpret linorder le lt by fact
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

dl-local-setup (Comparator-Generator.register-foreign-comparator @{typ unit}
  @{term comparator-unit}
  @{thm comparator-unit})
}
dl-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) = (≤)
    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 ≤ 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 ?thesis by 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)
      qed
      qed
show \text{?case}
  by (cases ys, simp-all)
next
case (Cons x xs yys) note IH = this
  thus \text{?case}
proof (cases yys)
  case Nil
  thus \text{?thesis} by auto
next
case (Cons y ys)
  show \text{?thesis unfolding Cons}
    using IH[\text{of ys}]
    by (cases x y rule: linorder-cases, auto)
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 (\text{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 \text{-} comparator-of) xy1 xy2 = (xy1 \leq xy2)
      by (cases xy1, cases xy2, auto)
  qed
  show lt-of-comp (\text{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 \text{-} 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 (\text{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]: \(\forall x y :: \text{real}. \; \text{invert-order (compare } x \; y\) = 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

24
In the following, we define a generator which for a given datatype \((a', b', c', z')\) \textit{Equality-Generator.type} constructs an equality-test function of type \((a \Rightarrow a \Rightarrow \text{bool}) \Rightarrow (b \Rightarrow b \Rightarrow \text{bool}) \Rightarrow (c \Rightarrow c \Rightarrow \text{bool}) \Rightarrow (z \Rightarrow z \Rightarrow \text{bool}) \Rightarrow \text{bool}\). These functions are essential to synthesize conditional equality functions in the container framework, where a strict membership in the \textit{equal}-class must not be enforced.

\textbf{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.

\textbf{definition list-all-eq :: bool list \Rightarrow bool where}
\text{list-all-eq = list-all id}

\textbf{5.1 Improved Code for Non-Lazy Languages}

The following equations will eliminate all occurrences of \text{list-all-eq} in the generated code of the equality functions.

\textbf{lemma list-all-eq-unfold:}
\text{list-all-eq [] = True}
\text{list-all-eq [b] = b}
\text{list-all-eq (b1 \# b2 \# bs) = (b1 \& list-all-eq (b2 \# bs))}
\textbf{unfolding list-all-eq-def by auto}

\textbf{lemma list-all-eq: list-all-eq bs \iff (\forall b \in \text{set. bs. b})}
\textbf{unfolding list-all-eq-def list-all-iff by auto}

\textbf{5.2 Partial Equality Property}

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

\textbf{type-synonym 'a equality = 'a \Rightarrow 'a \Rightarrow \text{bool}}

\textbf{definition pequality :: 'a equality \Rightarrow 'a \Rightarrow \text{bool where}}
\text{pequality aeq x \iff (\forall y. aeq x y \iff x = y)}

\textbf{lemma pequalityD: pequality aeq x \Rightarrow aeq x y \iff x = y}
\textbf{unfolding pequality-def by auto}

\textbf{lemma pequalityI: (\forall y. aeq x y \iff x = y) \Rightarrow pequality aeq x}
\textbf{unfolding pequality-def by auto}

\textbf{5.3 Global equality property}

\textbf{definition equality :: 'a equality \Rightarrow \text{bool where}}
\[ \text{equality aeq} \iff (\forall x. \text{pequality aeq } x) \]

**Lemma equalityD2:** equality aeq \(\implies\) pequality aeq x

**Unfolding equality-def by blast**

**Lemma equalityI2:** \((\forall x. \text{pequality aeq } x) \implies\) equality aeq

**Unfolding equality-def by blast**

**Lemma equalityD:** equality aeq \(\implies\) aeq x y \(\iff\) x = y

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

**Lemma equalityI:** \((\forall x y. \text{aeq } x y \iff x = y) \implies\) equality aeq

**By (intro equalityI2 pequalityI)**

**Lemma equality-imp-eq:**

\[\text{equality aeq} \implies\] 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. \(\text{int, integer, nat, char, bool, unit, sum, option, list, and prod}\). 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 ⇒ hashcode list ⇒ 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 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, sort) => (n, sort)) i end
    val thy' = Class.instantiation ([dtyp-name],vs,sort) thy
      |> Class.prove-instantiation-exit (fn ctx => countable-tac ctx 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 = (\_ :: 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

\[
\text{datatype } 'a \text{ bintree } = \text{BEmpty } | \text{BNode } 'a \text{ bintree } 'a 'a \text{ bintree}
\]

derive compare-order bintree
derive countable bintree
derive equality bintree
derive hashable bintree

9.3 Using Other datatypes

\[
\text{datatype } \text{nat-list-list } = NNil | CCons \text{ nat list } \times \text{rat option nat-list-list}
\]

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

\[
\text{datatype } 'a \text{ mtree } = \text{MEmpty } | \text{MNode } 'a 'a \text{ mtree-list and}
'a \text{ mtree-list } = \text{MNil } | \text{MCons } 'a \text{ mtree } 'a \text{ mtree-list}
\]

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

\[
\text{datatype } 'a \text{ tree } = \text{Empty } | \text{Node } 'a 'a \text{ tree list}
\]
\[
\text{datatype } 'a \text{ ttree } = \text{TEmpty } | \text{TNode } 'a 'a \text{ ttree list tree}
\]

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

9.6 Examples from IsaFoR

\[
\text{datatype } ('f,'v) \text{ term } = \text{Var } 'v | \text{Fun } 'f ('f,'v) \text{ term list}
\]
\[
\text{datatype } ('f,'l) \text{ lab } =
\text{Lab } ('f,'l) \text{ lab } 'l
| \text{FunLab } ('f,'l) \text{ lab } ('f,'l) \text{ lab list}
| \text{UnLab } 'f
\]

30
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

