Deriving class instances for datatypes.*

Christian Sternagel and René Thiemann

August 16, 2018

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 ................................................. 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.
1http://cl-informatik.uibk.ac.at/software/ceta
4 Generating Comparators
4.1 Lexicographic combination of order ............................. 14
4.2 Improved code for non-lazy languages .......................... 14
4.3 Pointwise properties for equality, symmetry, and transitivity ... 14
4.4 Separate properties of comparators ............................... 16
4.5 Auxiliary Lemmas for Comparator Generator ..................... 18
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 ........... 22
4.10 Compare Instance for Rational Numbers .......................... 24
4.11 Compare Instance for Real Numbers ............................... 24

5 Checking Equality Without "=
5.1 Improved Code for Non-Lazy Languages ......................... 25
5.2 Partial Equality Property ......................................... 25
5.3 Global equality property ....................................... 25
5.4 The Generator ..................................................... 26
5.5 Defining Equality-Functions for Common Types .................. 26

6 Generating Hash-Functions
6.1 Improved Code for Non-Lazy Languages ......................... 27
6.2 The Generator ..................................................... 27
6.3 Defining Hash-Functions for Common Types ..................... 27

7 Countable Datatypes
7.1 Installing the tactic ............................................... 28

8 Loading Existing Derive-Commands

9 Examples
9.1 Rational Numbers .................................................. 29
9.2 A Datatype Without Nested Recursion .......................... 30
9.3 Using Other datatypes .......................................... 30
9.4 Mutual Recursion .................................................. 30
9.5 Nested recursion .................................................. 30
9.6 Examples from IsaFoR .......................................... 30
9.7 A Complex Datatype .............................................. 31

10 Acknowledgements

2
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 ∈ set (y # z # ys) = (x = y ∨ x ∈ set (z # ys))
  x ∈ set ([y]) = (x = y)
  x ∈ set [] = False
Ball (set []) P = 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 ⇒ bool

In the following, we provide the obvious definitions how to switch be-
tween 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 P else Q) 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 P 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
by unfold-locales (auto split: if-splits simp: comparator-of-def)

end

3.2 Compare

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

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

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

lemma compare-Eq-is-eq [simp]:
compare x y = Eq ↔ 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:


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

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

thm two-comparisons-into-compare
end

ML-file compare-code.ML

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

export-code 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 provably 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-comp-ins :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a, 'b) rbt
where
  rbt-comp-ins f k v RBT-Impl.Empty = Branch RBT-Impl.R RBT-Impl.Empty k v
  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 (f k y v) r)
  rbt-comp-ins f k v (Branch RBT-Impl.R l x y r) = (case c k x of
    Lt ⇒ Branch RBT-Impl.R (rbt-comp-ins f k v l) x y r
   | Gt ⇒ Branch RBT-Impl.R l x y (rbt-comp-ins f k v r)
   | Eq ⇒ Branch RBT-Impl.R l x (f k y v) r)

definition rbt-comp-insert-with-key :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ 'a ⇒ '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
and rbt-comp-del-from-right :: 'a ⇒ ('a,'b) rbt ⇒ 'a ⇒ 'b ⇒ ('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 x 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 v rt) y s b = balance-left (rbt-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 a)

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 ⇒ ('b ⇒ ('b ⇒ 'b)))) ⇒ ('a × 'b) list |
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)
function comp-sunion-with :: ('a ⇒ 'b ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) 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 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)
\[ \text{list} \Rightarrow (a \times b) \text{ list} \]

\textbf{where}

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

\[
\begin{cases}
\text{Lt} \Rightarrow \text{comp-sinter-with } f ((k, v) \# as) bs \\
\text{Gt} \Rightarrow \text{comp-sinter-with } f as ((k', v') \# bs) \\
\text{Eq} \Rightarrow (k, f k v v') \# \text{comp-sinter-with } f as bs
\end{cases}
\]

\text{by pat-completeness auto}

\textbf{termination by lexicographic-order}

\textbf{definition rbt-comp-union-with-key} :: \((a \Rightarrow b \Rightarrow b) \Rightarrow (a, 'b) rbt \Rightarrow (a, 'b) rbt \Rightarrow (a, 'b) rbt \Rightarrow (a, 'b) rbt\)

\textbf{where}

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

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

\textbf{definition rbt-comp-inter-with-key} :: \((a \Rightarrow b \Rightarrow b \Rightarrow b) \Rightarrow (a, 'b) rbt \Rightarrow (a, 'b) rbt \Rightarrow (a, 'b) rbt \Rightarrow (a, 'b) rbt\)

\textbf{where}

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

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

\textbf{context}

\textbf{assumes c: comparator c

begin

\textbf{lemma rbt-comp-lookup: rbt-comp-lookup = ord.rbt-lookup (lt-of-comp c)\n
proof (intro ext)\n
fix k and t :: ('a,'b)rbt\n
show rbt-comp-lookup t k = ord.rbt-lookup (lt-of-comp c) t k

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

(auto split: order.splits)

qed}
lemma rbt-comp-ins: 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

lemma rbt-comp-insert-with-key: rbt-comp-insert-with-key = ord.rbt-insert-with-key (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-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 \( \text{rbt-comp-map-entry} f k t = \text{ord.rbt-map-entry} (\text{lt-of-comp} c) f k t \)
by \( \text{(induct} t, \text{unfold rbt-comp-map-entry.simps ord.rbt-map-entry.simps comparator.two-comparisons-into-case-order[OF c]} \)\)
(auto split: order.splits)
qed

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

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

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

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

lemmas \( \text{rbt-comp-simps} = \)
\( \text{rbt-comp-insert} \)
\( \text{rbt-comp-lookup} \)
\( \text{rbt-comp-delete} \)
\( \text{rbt-comp-bulkload} \)
\( \text{rbt-comp-map-entry} \)
\( \text{rbt-comp-union-with-key} \)
\( \text{rbt-comp-inter-with-key} \)
end
end
end

13
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** \texttt{trans-orderI}:

\[(x \neq Gt \implies y \neq Gt \implies z \neq Gt \land ((x = Lt \lor y = Lt) \implies z = Lt)) \implies\]

\text{trans-order} \ x \ y \ z

\text{by} (\text{simp add: trans-order-def})

**Lemma** \texttt{trans-orderD}:

\text{assumes} \ \text{trans-order} \ x \ y \ z \ \text{and} \ x \neq Gt \ \text{and} \ y \neq Gt

\text{shows} \ z \neq Gt \ \text{and} \ x = Lt \lor y = Lt \implies z = Lt

\text{using} \ \text{assms} \ \text{by} (\text{auto simp: trans-order-def})

**Lemma** \texttt{All-less-Suc}:

\[\forall i < \text{Suc} \ x. \ P i \longleftrightarrow P 0 \land (\forall i < x. \ P (\text{Suc} i))\]

\text{using} \ \text{less-Suc-eq-0-disj} \ \text{by force}

**Lemma** \texttt{comp-lex-trans}:

\text{assumes} \ \text{length} \ xs = \text{length} \ ys

\text{and} \ \forall i < \text{length} \ zs. \ \text{trans-order} \ (xs ! i) \ (ys ! i) \ (zs ! i)

\text{shows} \ \text{trans-order} \ (\text{comp-lex} \ xs) \ (\text{comp-lex} \ ys) \ (\text{comp-lex} \ zs)

\text{using} \ \text{assms}

\text{proof} (\text{induct} \ xs \ ys \ zs \ \text{rule: list-induct3})

\text{case} (\text{Cons} \ x \ xs \ y \ ys \ z \ zs)

\text{then show} \ ?\text{case}

\text{by} (\text{intro} \ \text{trans-orderI})

\text{(cases} \ x \ y \ z \ \text{rule: order.exhaust [case-product order.exhaust order.exhaust], auto simp: All-less-Suc dest: trans-orderD})

\text{qed} (\text{simp add: trans-order-def})

**Lemma** \texttt{comp-lex-sym}:

\text{assumes} \ \text{length} \ xs = \text{length} \ ys

\text{and} \ \forall i < \text{length} \ ys. \ \text{invert-order} \ (xs ! i) = ys ! i

\text{shows} \ \text{invert-order} \ (\text{comp-lex} \ xs) = \text{comp-lex} \ ys

\text{using} \ \text{assms} \ \text{by} (\text{induct} \ xs \ ys \ \text{rule: list-induct2, simp, case-tac} \ x \ \text{fastforce+})

\text{declare} \ \text{comp-lex.simps [simp del]}

**Definition** \texttt{peq-comp} :: 'a comparator \Rightarrow 'a \Rightarrow bool

\text{where}

\text{peq-comp} \ acomp \ x \leftarrow (\forall y. \ acomp \ x \ y = Eq \longleftrightarrow x = y)

**Lemma** \texttt{peq-compD}: \text{peq-comp} \ acomp \ x \Rightarrow acomp \ x \ y = Eq \longleftrightarrow x = y

\text{unfolding} \ \text{peq-comp-def} \ \text{by auto}

**Lemma** \texttt{peq-compI}: (\forall y. \ acomp \ x \ y = Eq \longleftrightarrow x = y) \Rightarrow \text{peq-comp} \ acomp \ x

\text{unfolding} \ \text{peq-comp-def} \ \text{by auto}

**Definition** \texttt{psym-comp} :: 'a comparator \Rightarrow 'a \Rightarrow bool

\text{where}

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

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

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

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

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

4.4 Separate properties of comparators

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

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

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

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

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

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

definition \text{sym-comp} :: 'a \ \text{comparator} \Rightarrow \text{bool} \ \text{where}  
\text{sym-comp acomp} \leftarrowarrow (\forall \ x. \ \text{psym-comp acomp x})
lemma sym-compD2:
  sym-comp acomp ⇒ psym-comp acomp x
unfolding sym-comp-def by blast

lemma sym-compI2: (∀ x. psym-comp acomp x) ⇒ sym-comp acomp
unfolding sym-comp-def by blast

lemma eq-compD: eq-comp acomp ⇒ acomp x y = Eq ↔ x = y
  by (rule peq-compD [OF eq-compD2])

lemma eq-compI: (∀ x y. acomp x y = Eq ↔ x = y) ⇒ eq-comp acomp
  by (intro eq-compI2 peq-compI)

lemma trans-compD: trans-comp acomp ⇒ trans-order (acomp x y) (acomp y z) (acomp x z)
  by (rule ptrans-compD [OF trans-compD2])

lemma trans-compI: (∃ x y z. trans-order (acomp x y) (acomp y z) (acomp x z)) ⇒ trans-comp acomp
  by (intro trans-compI2 ptrans-compI)

lemma sym-compD:
  sym-comp acomp ⇒ invert-order (acomp x y) = (acomp y x)
  by (rule psym-compD [OF sym-compD2])

lemma sym-compI: (∀ x y. invert-order (acomp x y) = (acomp y x)) ⇒ sym-comp acomp
  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 (acomp x y) = acompp y x
    using sym-compD [OF sym-compD] .
    { assume acompp x y = Eq
      with eq-compD [OF eq-compD]
      show x = y by blast
    }
    { assume acompp x y = Lt and acompp y z = Lt
      with trans-orderD [OF trans-compD [OF trans-compD]] , of x y z
      show acompp x z = 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:
  length [] = 0
  length (x # xs) = Suc (length xs)
  (x # xs)!0 = x
  (x # xs)!Suc n = xs!n
  by auto

4.6 The Comparator Generator

ML-file comparator-generator.ML

end

4.7 Compare Generator

theory Compare-Generator
imports Comparator-Generator Compare
begin

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

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

• derive compare prod creates an instance prod :: (compare, compare) compare.

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

• derive linorder prod creates an instance prod :: (linorder, linorder) linorder.

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

lemma linorder-axiomsD: assumes class.linorder le lt
  shows
      lt x y = (le x y ∧ ¬ le y x) (is ?a)
      le x x (is ?b)
      le x y ⇒ le y z ⇒ le x z (is ?c1 ⇒ ?c2 ⇒ ?c3)
      le x y ⇒ le y x ⇒ x = y (is ?d1 ⇒ ?d2 ⇒ ?d3)
      le x y ∨ le y x (is ?e)
proof
  interpret linorder le lt by fact
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 not used to instantiate linorder.

  derive compare sum list prod option

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

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

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

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

lemma comparator-bool: comparator comparator-bool
proof
  fix x y z :: bool
  show invert-order (comparator_bool x y) = comparator_bool y x by (cases x, (cases y, auto)+)
  show comparator_bool x y = Eq \implies x = y by (cases x, (cases y, auto)+)
  show comparator_bool x y = Lt \implies comparator_bool y z = Lt \implies comparator_bool x z = Lt
    by (cases x, (cases y, auto), cases y, (cases z, auto)+)
qed

local-setup ⟨⟨ Comparator_Generator.register-foreign-comparator @\{typ unit\}
  @\{term comparator-unit\}
  @\{thm comparator-unit\} ⟩⟩

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 ?thesis by (cases ys, simp-all)
  next
    case (Cons x xs yys)
    note IH = this
    thus ?case
    proof (cases yys)
      case Nil
      thus ?thesis by auto
    next
      case (Cons y ys)
      show ?thesis unfolding Cons
        using IH[of ys]
        by (cases x y rule: linorder-cases, auto)
  qed
  qed

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
show \( \text{?case} \)  
\hspace{1em} \text{by (cases } y \text{, simp-all) } 
\text{next} 
\hspace{1em} \text{case } (\text{Cons } x \text{ xs } yys) \text{ note } IH = \text{this} 
\hspace{1em} \text{thus } \text{?case} 
\hspace{1em} \text{proof (cases } yys) 
\hspace{2em} \text{case } Nil 
\hspace{3em} \text{thus } \text{?thesis by auto} 
\text{next} 
\hspace{1em} \text{case } (\text{Cons } y \text{ ys}) 
\hspace{1em} \text{show } \text{?thesis unfolding } Cons 
\hspace{2em} \text{using } IH[\text{of } yys] 
\hspace{3em} \text{by (cases } x \text{ y rule: } \text{linorder-cases, auto)} 
\text{qed} 
\text{qed} 
\text{qed} 
\text{qed} 
\text{instance } \text{prod :: (compare-order, compare-order) compare-order} 
\text{proof} 
\hspace{1em} \text{note } [\text{simp}] = \text{le-of-comp-def lt-of-comp-def comparator-of-def} 
\hspace{1em} \text{show } \text{le-of-comp (compare :: } (\text{'a,}'b)\text{prod comparator)} = (\leq) 
\hspace{2em} \text{unfolding } compare\text{-prod-def compare-is-comparator-of} 
\hspace{3em} \text{proof (intro ext)} 
\hspace{4em} \text{fix } xy1 \text{ xy2 :: } (\text{'a,}'b)\text{prod} 
\hspace{5em} \text{show } \text{le-of-comp (comparator-prod comparator-of comparator-of) } xy1 \text{ xy2} = (xy1 \leq xy2) 
\hspace{6em} \text{by (cases } xy1, \text{ cases } xy2, \text{ auto)} 
\text{qed} 
\text{show } \text{lt-of-comp (compare :: } (\text{'a,}'b)\text{prod comparator)} = (<) 
\hspace{2em} \text{unfolding } compare\text{-prod-def compare-is-comparator-of} 
\hspace{3em} \text{proof (intro ext)} 
\hspace{4em} \text{fix } xy1 \text{ xy2 :: } (\text{'a,}'b)\text{prod} 
\hspace{5em} \text{show } \text{lt-of-comp (comparator-prod comparator-of comparator-of) } xy1 \text{ xy2} = (xy1 < xy2) 
\hspace{6em} \text{by (cases } xy1, \text{ cases } xy2, \text{ auto)} 
\text{qed} 
\text{qed} 
\text{instance } \text{option :: (compare-order) compare-order} 
\text{proof} 
\hspace{1em} \text{note } [\text{simp}] = \text{le-of-comp-def lt-of-comp-def comparator-of-def} 
\hspace{1em} \text{show } \text{le-of-comp (compare :: } \text{'a option comparator)} = (\leq) 
\hspace{2em} \text{unfolding } compare\text{-option-def compare-is-comparator-of} 
\hspace{3em} \text{proof (intro ext)} 
\hspace{4em} \text{fix } xy1 \text{ xy2 :: } \text{'a option} 
\hspace{5em} \text{show } \text{le-of-comp (comparator-option comparator-of) } xy1 \text{ xy2} = (xy1 \leq xy2) 
\hspace{6em} \text{by (cases } xy1, \text{ (cases } xy2, \text{ auto split: if-splits)+)} 
\text{qed} 
\text{qed}
show \texttt{lt-of-comp} (\texttt{compare :: 'a option comparator}) = (<)
unfolding \texttt{compare-option-def compare-is-comparator-of}
proof (intro ext)
  fix \texttt{xy1 \ xy2 :: 'a option}
  show \texttt{lt-of-comp} (\texttt{comparator-option comparator-of \ xy1 \ xy2} = (\texttt{xy1 < xy2})
    by (cases \texttt{xy1}, (cases \texttt{xy2}, auto split: if-splits)+)
qed

end

4.10 Compare Instance for Rational Numbers

theory \textit{Compare-Rat}
imports \textit{Compare-Generator H\textsc{ol.Rat}}
begin

derive (linorder) compare-order rat

end

4.11 Compare Instance for Real Numbers

theory \textit{Compare-Real}
imports \textit{Compare-Generator H\textsc{ol.Real}}
begin

derive (linorder) compare-order real

lemma \texttt{invert-order-compare-real[simp]: A x y :: real. invert-order (compare x y) = compare y x}
  by (simp add: comparator-of-def compare-is-comparator-of)

end

5 Checking Equality Without ”=”

theory \textit{Equality-Generator}
imports \\
  ../\textit{Generator-Aux} \\
  ../\textit{Derive-Manager}
begin

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

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

\texttt{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.

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

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

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

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

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

\texttt{5.2 Partial Equality Property}

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

\texttt{type-synonym \texttt{a equality = \texttt{a} \Rightarrow \texttt{a} \Rightarrow \texttt{bool}}}

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

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

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

\texttt{5.3 Global equality property}

\texttt{definition equality :: \texttt{\texttt{a equality} \Rightarrow \texttt{bool} where}
equality aeq \iff (\forall x. \text{pequality aeq } x)

lemma equalityD2: equality aeq \implies \text{pequality aeq } x
unfolding equality-def by blast

lemma equalityI2: (\land x. \text{pequality aeq } x) \implies \text{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: (\land x y. aeq x y \iff x = y) \implies \text{equality aeq }
by (intro equalityI2 pequalityI)

lemma equality-imp-eq:
  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
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.

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

The Generator

ML-file hash-generator.ML

end

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.

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
fan derive dtyp-name - thy =

let
val base-name = Long-Name.base-name dtyp-name
val _ = writeln (proving that datatype `base-name` is countable)
val sort = @ {sort countable}
val vs =
let val i = BNF-LFP-Compat.the-spec thy dtyp-name | > #1
in map (fn (n, _) => (n, sort)) i end
val thy' = Class.instantiation ([dtyp-name], vs, sort) thy
| > Class.prove-instantiation-exit (fn cxxt => countable-tac cxxt 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
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.
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 × rat option nat-list-list

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

9.4 Mutual Recursion

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

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

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

9.5 Nested recursion

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

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

9.6 Examples from IsaFoR

datatype ('f,'v) term = Var 'v | Fun 'f ('f,'v) term list
datatype ('f, 'l) lab =
   Lab ('f, 'l) lab 'l
| FunLab ('f, 'l) lab ('f, 'l) lab list
| UnLab 'f
9.7 A Complex Datatype

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

```haskell
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`.

```haskell
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

