Deriving class instances for datatypes.*

Christian Sternagel and René Thiemann

September 19, 2015

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 2

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

---

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

\(^1\)http://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

5 Checking Equality Without "="

5.1 Improved Code for Non-Lazy Languages ............................ 24
5.2 Partial Equality Property .................................................. 25
5.3 Global equality property .................................................. 25
5.4 The Generator .............................................................. 25
5.5 Defining Equality-Functions for Common Types .................. 26

6 Generating Hash-Functions

6.1 Improved Code for Non-Lazy Languages ............................ 26
6.2 The Generator .............................................................. 27
6.3 Defining Hash-Functions for Common Types ....................... 27

7 Countable Datatypes

7.1 Installing the tactic ......................................................... 27

8 Loading Existing Derive-Commands

9 Examples

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

10 Acknowledgements

1 Derive Manager

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

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: \( a = b \Rightarrow c = d \Rightarrow (a \land c) = (b \land d) \) 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, \((op <, op \leq)\), one can alternative use a comparator to define the linear order, which may deliver three possible outcomes when comparing two values.

datatype order = Eq | Lt | Gt

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

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

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

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

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

lemma comp-of-ords-of-le-lt[simp]: comp-of-ords (le-of-comp c) (lt-of-comp c) = c
  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: (\( \forall x y. \lt x y \Rightarrow \leq x y \)) \( \Rightarrow \) 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 ⇒ z = 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 Q else (if le x y then P else R)) = (case-order P Q R (comp x y))
(if lt x y then Q else (if lt y x then R else P)) = (case-order P Q R (comp x y))
(if lt x y then Q else (if x = y then P else R)) = (case-order P Q R (comp x y))
(if lt x y then Q else (if y = x then P else R)) = (case-order P Q R (comp x y))
(if x = y then P else (if lt y x then Q else R)) = (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
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 = op ≤ lt-of-comp comparator-of = op <
  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 = op ≤ lt-of-comp compare = op <

  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

end
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 \texttt{op} \texttt{\leq}, \texttt{op} \texttt{<"}, or \texttt{op} = are turned into one invocation of \texttt{compare}. The difference to a standard \texttt{code-unfold} is that here we change the code-equations where an additional sort-constraint on \texttt{compare-order} can be added. Otherwise, there would be no \texttt{compare}-function.

end

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

theory RBT-Compare-Order-Impl
imports
  Compare
  ~/src/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 \texttt{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

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

theory RBT-Comparator-Impl
imports ~~/src/HOL/Library/RBT-Impl Comparator
begin

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

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

context
fixes c :: 'a comparator
begin

primrec rbt-comp-lookup :: ('a, 'b) rbt ⇒ 'a ⇀ 'b
where
rbt-comp-lookup RBT-Impl.Empty k = None
| rbt-comp-lookup (Branch - l x y r) k =
  (case c k x of Lt ⇒ rbt-comp-lookup l k
  | Gt ⇒ rbt-comp-lookup r k
  | Eq ⇒ Some y)

fun rbt-comp-ins :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
where
rbt-comp-ins f k v RBT-Impl.Empty = Branch RBT-Impl.R RBT-Impl.Empty k
| 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 ⇒ ('a, 'b) rbt ⇒ ('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 z v rt) y s b = balance-left
(rbtt-comp-del x (Branch RBT-Impl.B lt z v rt)) 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)

function comp-sunion-with :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a × 'b) list ⇒ ('a × 'b) list

where
comp-sunion-with f ((k, v) # as) ((k', v') # bs) =
   (case 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

where
comp-sinter-with f ((k, v) # as) ((k', v') # bs) =

10
(case c k k af
  Lt ⇒ comp-sinter-with f ((k, v) ≠ as) bs
  | Gt ⇒ comp-sinter-with f as ((k', v') ≠ bs)
  | Eq ⇒ (k, f k v v') ≠ comp-sinter-with f as bs)
| comp-sinter-with f [] = []
| comp-sinter-with f - [] = []
by pat-completeness auto
termination by lexicographic-order

definition rbt-comp-union-with-key :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
where
  rbt-comp-union-with-key f t1 t2 =
  (case RBT-Impl.compare-height t1 t1 t2 t2
  of compare.EQ ⇒ rbtreeify (comp-union-with f (RBT-Impl.entries t1) (RBT-Impl.entries t2))
  | compare.LT ⇒ RBT-Impl.fold (rbt-comp-insert-with-key (λk v w. f k w v)) t1 t2
  | compare.GT ⇒ RBT-Impl.fold (rbt-comp-insert-with-key f) t2 t1)

definition rbt-comp-inter-with-key :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
where
  rbt-comp-inter-with-key f t1 t2 =
  (case RBT-Impl.compare-height t1 t1 t2 t2
  of compare.EQ ⇒ rbtreeify (comp-sinter-with f (RBT-Impl.entries t1) (RBT-Impl.entries t2))
  | compare.LT ⇒ rbtreeify (List.map-filter (λ(k, v). map-option (λw. (k, f k w v))) (rbt-comp-lookup t2 k) (RBT-Impl.entries t1))
  | compare.GT ⇒ rbtreeify (List.map-filter (λ(k, v). map-option (λw. (k, f k w v))) (rbt-comp-lookup t1 k) (RBT-Impl.entries t2)))

context
  assumes c: comparator c

begin

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

(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])
\begin{center}
\begin{verbatim}
(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
\end{verbatim}
\end{center}
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 sound-
ness of comparators. They are defined in a way that are combinable with
comp-lex.

lemma comp-lex-eq; comp-lex os = Eq ℓ→ (∀ 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:
\[(x \neq Gt \implies y \neq Gt \implies z \neq Gt \land ((x = Lt \lor y = Lt) \implies z = Lt)) \implies\]

**trans-order** \(x\ y\ z\)

by (simp add: trans-order-def)

**lemma** trans-orderD:

assumes trans-order \(x\ y\ z\) and \(x \neq Gt\) and \(y \neq Gt\)

shows \(z \neq Gt\) and \(x = Lt\lor y = Lt \implies z = Lt\)

using assms by (auto simp: trans-order-def)

**lemma** All-less-Suc:

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

using less-Suc-eq-0-disj by force

**lemma** comp-lex-trans:

assumes length \(xs\) = length \(ys\)

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

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

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)

(auto simp: All-less-Suc dest: trans-orderD)

qed (simp add: trans-order-def)

**declare** comp-lex.simps |simp del|

**definition** pse-comp :: 'a comparator \(\Rightarrow\)'a \(\Rightarrow\) bool

where

\[\text{pse-comp } \text{acomp } x \longleftrightarrow (\forall y. \text{acomp } x y = \text{Eq} \longleftrightarrow x = y)\]

**lemma** pse-compD: pse-comp \(\text{acomp } x\) \(\implies\) \(\text{acomp } x y = \text{Eq} \longleftrightarrow x = y\)

unfolding pse-comp-def by auto

**lemma** pse-compI: \((\land y. \text{acomp } x y = \text{Eq} \longleftrightarrow x = y) \implies \text{pse-comp } \text{acomp } x\)

unfolding pse-comp-def by auto

**definition** pse-comp :: 'a comparator \(\Rightarrow\)'a \(\Rightarrow\) bool

where

\[\text{pse-comp } \text{acomp } x \longleftrightarrow (\forall y. \text{invert-order } (\text{acomp } x y) = (\text{acomp } y x))\]

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

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

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

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

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

4.4 Separate properties of comparators
definition \(\text{eq-comp : 'a comparator \Rightarrow \'}\ a \Rightarrow \text{bool \ where}\)
\[\text{eq-comp \ acomp \ \rightleftharpoons (\forall \ x. \text{peq-comp \ acomp \ x})}\]

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

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

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

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

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

definition \(\text{sym-comp : 'a comparator \Rightarrow \'}\ a \Rightarrow \text{bool \ where}\)
\[\text{sym-comp \ acomp \ \rightleftharpoons (\forall \ x. \text{psym-comp \ acomp \ x})}\]

lemma sym-compD2:
sym-comp acomp \implies psym-comp acomp x

unfolding sym-comp-def by blast

lemma sym-compI2: (\forall x. psym-comp acomp x) \implies sym-comp acomp
unfolding sym-comp-def by blast

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

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

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

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

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

lemma sym-compI: (\forall x y. invert-order (acomp x y) = (acomp y x)) \implies 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) = acomp y x
using sym-compD [OF (sym-comp acomp)].
{
assume acomp x y = Eq
with eq-compD [OF (eq-comp acomp)]
show x = y by blast
}
{
assume acomp x y = Lt and acomp y z = Lt
with trans-orderD [OF trans-compD [OF (trans-comp acomp)], of x y z]
show acomp 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
le x x = (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

19
Show \( ?a \, ?b \, ?c_1 \Rightarrow ?c_2 \Rightarrow ?c_3 \, ?d_1 \Rightarrow ?d_2 \Rightarrow ?d_3 \, ?e \) by auto

QED

Named-theorems compare-simps simp theorems to derive compare = comparator-of

ML-file compare-generator.ML

End

4.8 Defining Comparators and Compare-Instances for Common Types

Theory Compare-Instances
Imports
  Compare-Generator
  ~/src/HOL/Library/Char-ord
Begin

For all of the following types, we define comparators and register them in the class compare: int, integer, nibble, 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, nibble, and char we just use their linear orders as comparators.

Derive \((\text{linorder})\) compare-order int integer nibble 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 ⇒ x = y by (cases x, (cases y, auto)+)
show comparator-bool x y = Lt ⇒ comparator-bool y z = Lt ⇒ 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
  ~~/src/HOL/Library/List-lexord
  ~~/src/HOL/Library/Product-Lexorder
  ~~/src/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) = op ≤`
    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`
  thus `?case` by (cases `ys`, simp-all)
  next
  case `(Cons x xs yys) note IH = this
  thus `?case` proof (cases `ys`
  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) = op <`
    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)`

22
show ?case
  by (cases ys, simp-all)

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

instance prod :: (compare-order, compare-order) compare-order
  proof
    note [simp] = le-of-comp-def lt-of-comp-def comparator-of-def
    show le-of-comp (compare :: (′a,′b) prod comparator) = op ≤
      unfolding compare-prod-def compare-is-comparator-of
    proof (intro ext)
      fix xy1 xy2 :: (′a,′b) prod
      show le-of-comp (comparator-prod comparator-of comparator-of) xy1 xy2 = (xy1 ≤ xy2)
        by (cases xy1, cases xy2, auto)
    qed
    show lt-of-comp (compare :: (′a,′b) prod comparator) = op <
      unfolding compare-prod-def compare-is-comparator-of
    proof (intro ext)
      fix xy1 xy2 :: (′a,′b) prod
      show lt-of-comp (comparator-prod comparator-of comparator-of) xy1 xy2 = (xy1 < xy2)
        by (cases xy1, cases xy2, auto)
    qed
  qed

instance option :: (compare-order) compare-order
  proof
    note [simp] = le-of-comp-def lt-of-comp-def comparator-of-def
    show le-of-comp (compare :: ′a option comparator) = op ≤
      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 ≤ xy2)
        by (cases xy1, (cases xy2, auto split: if-splits))
    qed
show \text{lt-of-comp} (\text{compare} :: \text{'a option comparator}) = \text{op <}

unfolding \text{compare-option-def compare-is-comparator-of}

proof (intro ext)
  fix xy1 xy2 :: \text{'a option}
  show \text{lt-of-comp} (\text{comparator-option comparator-of}) xy1 xy2 = (xy1 < xy2)
  by (cases xy1, (cases xy2, auto split: if-splits)+)
qed
qed
end

5 Checking Equality Without ":="

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

typedecl \text{('a,'b,'c,'z)type}

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

hide-type type

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

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

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.

lemma \text{list-all-eq-unfold:}
  \text{list-all-eq [] = True}
  \text{list-all-eq [b] = b}
  \text{list-all-eq \text{(b1 # b2 # bs) = (b1 \land list-all-eq \text{b2 # bs})}}

unfolding \text{list-all-eq-def}
by auto

24
lemma list-all-eq: list-all-eq bs \iff (\forall b \in \text{set} \ bs. \ b)
unfolding list-all-eq-def list-all-iff by auto

5.2 Partial Equality Property

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

type-synonym 'a equality = 'a \Rightarrow 'a \Rightarrow \text{bool}
definition pequality :: 'a equality \Rightarrow 'a \Rightarrow \text{bool} where
  pequality aeq x \iff (\forall y. \ aeq \ x \ y \iff \ x = \ y)
lemma pequalityD: pequality aeq x \Rightarrow aeq x y \iff x = y
  unfolding pequality-def by auto
lemma pequalityI: (\forall y. \ aeq \ x \ y \iff \ x = \ y) \Rightarrow \ pequality \ aeq \ x
  unfolding pequality-def by auto

5.3 Global equality property

definition equality :: 'a equality \Rightarrow \text{bool} where
  equality aeq \iff (\forall x. \ pequality \ aeq \ x)
lemma equalityD2: equality aeq \Rightarrow \ pequality \ aeq \ x
  unfolding equality-def by blast
lemma equalityI2: (\forall x. \ pequality \ aeq \ x) \Rightarrow equality \ aeq
  unfolding equality-def by blast
lemma equalityD: equality aeq \Rightarrow aeq x y \iff x = y
  by (rule pequalityD[OF equalityD2])
lemma equalityI: (\forall x y. \ aeq \ x \ y \iff \ x = \ y) \Rightarrow equality \ aeq
  by (intro equalityI2 pequalityI)
lemma equality-imp-eq:
  equality aeq \Rightarrow aeq = (op =)
  by (intro ext, auto dest: equalityD)
lemma eq-equality: equality (op =)
  by (rule equalityI, simp)
lemma equality-def: equality f = (f = op =)
  using equality-imp-eq eq-equality by blast

5.4 The Generator

ML-file equality-generator.ML
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. \textit{int}, \textit{integer}, \textit{nibble}, \textit{nat}, \textit{char}, \textit{bool}, \textit{unit}, \textit{sum}, \textit{option}, \textit{list}, and \textit{prod}. For types without type parameters, we use plain $\textit{op} =$, 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 nibble nat char bool unit
derive equality sum list prod option

end

6 Generating Hash-Functions

theory Hash-Generator
imports ..../Generator-Aux ..../Derive-Manager ../../../Collections/Lib/HashCode
begin

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

fun hash-combine :: hashcode list $\Rightarrow$ hashcode list $\Rightarrow$ hashcode where
hash-combine $\emptyset$ $\langle x \rangle$ = $x$
| hash-combine ($y \#$ $ys$) ($z \#$ $zs$) = $y \ast z +$ hash-combine $ys$ $zs$
| hash-combine -- = 0

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

6.1 Improved Code for Non-Lazy Languages

lemma hash-combine-unfold:
hash-combine $\emptyset$ $\langle x \rangle$ = $x$
hash-combine ($y \#$ $ys$) ($z \#$ $zs$) = $y \ast z +$ 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. \texttt{int}, \texttt{integer}, \texttt{nibble}, \texttt{nat}, \texttt{char}, \texttt{bool}, \texttt{unit}, \texttt{sum}, \texttt{option}, \texttt{list}, and \texttt{prod}. For types without type parameters, we use plain \texttt{hashcode}, and for the others we use generated ones.

\texttt{derive (hashcode) hash-code int integer nibble bool char unit nat}

\texttt{derive hash-code prod sum option list}

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

end

7 Countable Datatypes

theory Countable-Generator
imports
  ~/src/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 \texttt{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
  end
```
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 ctxt => countable-tac ctxt 1)
val _ = writeln (registered ` base-name ` in class countable)
in thy' end

in 
  Derive-Manager.register-derive countable register datatypes is class countable
derive end

end

8 Loading Existing Derive-Commands

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

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

  print-derives

end

9 Examples

theory Derive-Examples
imports
  Derive
  Comparator-Generator/Compare-Order-Instances
  Equality-Generator/Equality-Instances
  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.

\[ \text{derive (linorder) compare-order rat} \]

Use \( op = \) as equality function.

\[ \text{derive (eq) equality rat} \]

First manually define a hashcode function.

\[ \text{instantiation rat :: hashable} \]

\[ \text{begin} \]
\[ \text{definition def-hashmap-size} = (\lambda r :: \text{rat itself}. 10) \]
\[ \text{definition hashcode (r :: rat)} = \text{hashcode (quotient-of r)} \]
\[ \text{instance} \]
\[ \text{by (intro-classes)(simp-all add: def-hashmap-size-rat-def)} \]
\[ \text{end} \]

And then register it at the generator.

\[ \text{derive (hashcode) hash-code rat} \]

9.2 A Datatype Without Nested Recursion

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

\[ \text{derive compare-order bintree} \]
\[ \text{derive countable bintree} \]
\[ \text{derive equality bintree} \]
\[ \text{derive hashable bintree} \]

9.3 Using Other datatypes

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

\[ \text{derive compare-order nat-list-list} \]
\[ \text{derive countable nat-list-list} \]
\[ \text{derive (eq) equality nat-list-list} \]
\[ \text{derive hashable nat-list-list} \]

9.4 Mutual Recursion

\[ \text{datatype} \ 'a \ mtree = \text{MEmpty} | \text{MNode} 'a \ 'a \ mtree-list \text{ and} \]
\[ 'a \ mtree-list = \text{MNil} | \text{MCons 'a mtree 'a mtree-list} \]
derive compare-order mtree mtree-list
derive countable mtree mtree-list
derive hashable mtree mtree-list

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

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

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

9.6 Examples from IsaFoR
datatype ("f","v") term = Var "v" | Fun "f" ("f","v") term list
datatype ("f","l") lab =
  Lab ("f","l") lab "l"
  | FunLab ("f","l") lab ("f","l") lab list
  | UnLab "f"
  | Sharp ("f","l") lab
derive compare-order term lab
derive countable term lab
derive equality term lab
derive hashable term lab

9.7 A Complex Datatype
The following datatype has nested and mutual recursion, and uses other datatypes.
datatype ("a","b") complex =
  C1 nat 'a tree × 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 \texttt{complex2} we only derive \texttt{compare} which is not a subclass of \texttt{linorder}. The instance will be \texttt{complex2 :: (compare, compare) compare}, i.e., again the argument types have to be in class \texttt{compare}.

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

\begin{verbatim}
derive compare-order complex
derive compare complex2
derive linorder complex2
derive countable complex complex2
derive equality complex
derive hashable complex complex2
\end{verbatim}

\section{Acknowledgements}

We thank
\begin{itemize}
\item Lukas Bulwahn and Brian Huffman for the discussion on a generic derive command.
\item Jasmin Blanchette for providing the tactic for countability for BNF-based datatypes.
\item Jasmin Blanchette and Dmitriy Traytel for adjusting the Isabelle/ML interface of the BNF-based datatypes.
\item Alexander Krauss for telling us to avoid the function package for this task.
\item Peter Lammich for the inspiration of developing a hash-function generator.
\item Andreas Lochbihler for the inspiration of developing generators for the container framework.
\item Christian Urban for his cookbook on Isabelle/ML.
\item Stefan Berghofer, Florian Haftmann, Cezary Kaliszyk, Tobias Nipkow, and Makarius Wenzel for their explanations on several Isabelle related questions.
\end{itemize}
References

