Deriving class instances for datatypes.*

Christian Sternagel and René Thiemann

May 27, 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 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 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

*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 ...... 20

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.

\textbf{derive (param) sort datatype} calls the hook for deriving \textit{sort} (that may depend on the optional \textit{param}) on \textit{datatype} (if such a hook is registered).

E.g., \textbf{derive compare-order list} will derive a comparator for \textit{datatype list} which is also used to define a linear order on \textit{lists}.

There is also the diagnostic command \textbf{print-derives} that shows the list of currently registered hooks.

ML-file \textit{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.

\textbf{theory Generator-Aux}

\textbf{imports}
\hspace{1em} \textit{Main}

\textbf{begin}

ML-file \textit{bnf-access.ML}

ML-file \textit{generator-aux.ML}

\textbf{lemma in-set-simps:}
\begin{align*}
x \in \text{set } \{y \# z \# ys\} &= (x = y \lor x \in \text{set } \{z \# ys\}) \\
x \in \text{set } \{y\} &= (x = y) \\
x \in \text{set } [] &= \text{False} \\
\text{Ball } (\text{set } []) P &= \text{True} \\
\text{Ball } (\text{set } [x]) P &= P x \\
\text{Ball } (\text{set } (x \# y \# zs)) P &= (P x \land \text{Ball } (\text{set } (y \# zs)) P)
\end{align*}

\textbf{by auto}
Lemma conj-weak-cong: \( a = b \implies c = d \implies (a \land c) = (b \land d) \) 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, \((\text{op} <, \text{op} \leq)\), one can alternatively 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 \Rightarrow 'a \Rightarrow order

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

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

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

definition comp-of-ords :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow '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: \( \exists x y. \, \text{lt} x y \implies \text{le} x y \) \implies \text{le-of-comp} (\text{comp-of-ords} \text{le} \text{lt}) = \text{le}
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 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))
(if lt x y then Q else (if lt x y then Q else P)) = (case-order P Q R (comp x y))
(if lt x y then Q else (if lt x y then Q else P)) = (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))
(if x = y then P else (if le x y then Q else R)) = (case-order P Q R (comp x y))
(if x = y then P else (if le x y then Q else R)) = (case-order P Q R (comp x y))
(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 \(\text{linorder}\). 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 \iff 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 \leq 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 \leq 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
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 \( \text{op} \leq, \text{op} < \), or \( \text{op} = \) 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 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
  ~/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 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 ofLt ⇒ 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 ⇒ 'b) rbt ⇒ ('a, 'b) rbt
where
  rbt-comp-ins f k v RBT-Impl.Empty = Branch RBT-Impl.R RBT-Impl.Empty k
  f RBT-Impl.Empty 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 ⇒ ('a, 'b) rbt
and
\[\text{rbt-comp-del-from-right} :: 'a \Rightarrow ('a,'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a,'b) \Rightarrow ('a,'b)\]
and
\[\text{rbt-comp-del} :: 'a \Rightarrow ('a,'b) \Rightarrow ('a,'b)\]
where
\[\text{rbt-comp-del} x \text{ RBT-Impl.Empty} = \text{ RBT-Impl.Empty}\]
\[\text{rbt-comp-del} x (\text{Branch - a} y s b) =
\begin{align*}
\text{LT} & \Rightarrow \text{rbt-comp-del-from-left} x a y s b \\
\text{GT} & \Rightarrow \text{rbt-comp-del-from-right} x a y s b \\
\text{EQ} & \Rightarrow \text{combine} a b
\end{align*}\]
\[\text{rbt-comp-del-from-left} x \text{ RBT-Impl.} (\text{Branch - a} y s b) =
\begin{align*}
\text{LT} & \Rightarrow \text{balance-left} (\text{rbt-comp-del} x (\text{Branch - a} y s b)) y s b \\
\text{GT} & \Rightarrow \text{balance-right} a y s (\text{rbt-comp-del} x (\text{Branch - a} y s b))
\end{align*}\]
\[\text{rbt-comp-del-from-right} x \text{ RBT-Impl.} (\text{Branch - a} y s b) =
\begin{align*}
\text{LT} & \Rightarrow \text{balance-left} (\text{rbt-comp-del} x (\text{Branch - a} y s b)) y s b \\
\text{GT} & \Rightarrow \text{balance-right} a y s (\text{rbt-comp-del} x (\text{Branch - a} y s b))
\end{align*}\]
definition \text{rbt-comp-delete} k t = \text{paint} \text{ RBT-Impl.} (\text{rbt-comp-del} k t)
definition \text{rbt-comp-bulkload} xs = \text{foldr} (\lambda (k, v). \text{rbt-comp-insert} k v) xs \text{ RBT-Impl.Empty}
primrec
\[\text{rbt-comp-map-entry} :: 'a \Rightarrow ('b \Rightarrow ('c \Rightarrow 'd)) \Rightarrow ('e \Rightarrow 'd) \Rightarrow ('a \Rightarrow ('b \Rightarrow ('c \Rightarrow 'd)))\]
where
\[\text{rbt-comp-map-entry} k f \text{ RBT-Impl.Empty} = \text{ RBT-Impl.Empty}\]
\[\text{rbt-comp-map-entry} k f (\text{Branch cc lt x v rt}) =
\begin{align*}
\text{LT} & \Rightarrow \text{Branch} cc (\text{rbt-comp-map-entry} k f \text{ lt}) x v rt \\
\text{GT} & \Rightarrow \text{Branch} cc lt x v (\text{rbt-comp-map-entry} k f \text{ rt}) \\
\text{EQ} & \Rightarrow \text{Branch} cc lt x (f v) rt
\end{align*}\]
definition \text{comp-sunion-with} :: ((k, v) \Rightarrow 'a \times 'b) \Rightarrow ((k', v') \Rightarrow 'a \times 'b) \Rightarrow ((k, v) \Rightarrow 'a \times 'b) \Rightarrow ((k', v') \Rightarrow 'a \times 'b)
by \text{pat-completeness auto}
termination by \text{lexicographic-order}
definition \text{comp-sinter-with} :: ((k, v) \Rightarrow 'a \times 'b) \Rightarrow ((k', v') \Rightarrow 'a \times 'b) \Rightarrow ((k, v) \Rightarrow 'a \times 'b) \Rightarrow ((k', v') \Rightarrow 'a \times 'b)
by \text{pat-completeness auto}
\(\text{(case } c \ k \ k' \ a f\)

\(\text{Lt} \Rightarrow \text{comp-sinter-with } f ((k, v) \neq \text{ as}) \text{ bs}\)

\(\text{Gt} \Rightarrow \text{comp-sinter-with } f \text{ as } ((k', v') \neq \text{ bs})\)

\(\text{Eq} \Rightarrow (k, f k v v') \neq \text{ comp-sinter-with } f \text{ as bs}\)

\(\text{comp-sinter-with } f [\cdot] = \cdot\)

\(\text{comp-sinter-with } f \cdot = \cdot\)

by pat-completeness auto

termination by lexicographic-order

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

where

\(\text{rbt-comp-union-with-key } f \ t1 \ t2 =\)

(case \(\text{RBT-Impl.compare-height } t1 \ t1 \ t2 \ t2\)

of compare.EQ \Rightarrow \text{rbtreeify} (\text{comp-sunion-with } f (\text{RBT-Impl.entries } t1)) (\text{RBT-Impl.entries } t2))

| \text{compare.LT} \Rightarrow \text{RBT-Impl.fold} (\text{rbt-comp-insert-with-key} (\lambda k v w. f k w v)) \ t1 \ t2

| \text{compare.GT} \Rightarrow \text{RBT-Impl.fold} (\text{rbt-comp-insert-with-key } f \ t2 \ t1)

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

where

\(\text{rbt-comp-inter-with-key } f \ t1 \ t2 =\)

(case \(\text{RBT-Impl.compare-height } t1 \ t1 \ t2 \ t2\)

of compare.EQ \Rightarrow \text{rbtreeify} (\text{comp-sinter-with } f (\text{RBT-Impl.entries } t1)) (\text{RBT-Impl.entries } t2))

| \text{compare.LT} \Rightarrow \text{rbtreeify} (\text{List.map-filter} (\lambda (k, v). \text{map-option} (\lambda w. (k, f k v w))) (\text{rbt-comp-lookup } t2 \ k)) (\text{RBT-Impl.entries } t1))

| \text{compare.GT} \Rightarrow \text{rbtreeify} (\text{List.map-filter} (\lambda (k, v). \text{map-option} (\lambda w. (k, f k w v))) (\text{rbt-comp-lookup } t1 \ k)) (\text{RBT-Impl.entries } t2))

context

assumes c: comparator c

begin

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

proof (intro ext)

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

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

by (induct t, unfold \text{rbt-comp-lookup.simps ord.rbt-lookup.simps}

comparator.two-comparisons-into-case-order[OF c])

(auto split: order.splits)

qed

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

proof (intro ext)

fix f k v and t :: ('a, 'b)\text{ rbt}
show $\text{rbt-comp-ins } f \ k \ v \ t = \text{ord.rbt-ins } (\text{lt-of-comp } c) \ f \ k \ v \ t$
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 $\text{rbt-comp-insert-with-key: } rbt\text{-comp-insert-with-key} = \text{ord.rbt-insert-with-key} \ (\text{lt-of-comp } c)$
unfolding rbt-comp-insert-with-key-def[abs-def] ord.rbt-insert-with-key-def[abs-def]
unfolding rbt-comp-ins ..

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

lemma $\text{rbt-comp-del: } rbt\text{-comp-del} = \text{ord.rbt-del} \ (\text{lt-of-comp } c)$
proof 
fix $k \ a \ b$ and $s \ t :: (\'a,\'b)\text{rbt}$
have $\text{rbt-comp-del-from-left } k \ t \ a \ b \ s \ = \text{ord.rbt-del-from-left} \ (\text{lt-of-comp } c) \ k \ t \ a \ b \ s$
\text{rbt-comp-del-from-right } k \ t \ a \ b \ s \ = \text{ord.rbt-del-from-right} \ (\text{lt-of-comp } c) \ k \ t \ a \ b \ s$
\text{rbt-comp-del } k \ t \ = \text{ord.rbt-del} \ (\text{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
\text{rbt-comp-del-simps ord.rbt-del.simps}
\text{rbt-comp-del-from-left.simps ord.rbt-del-from-left.simps}
\text{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 \text{thesis} by (intro ext)
qed

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

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

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

qed

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

qed

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

qed

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

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

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

4 Generating Comparators

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

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

In the following, we define a generator which for a given datatype ('a, 'b, 'c, 'z) Comparator-Generator\text{.\textit{type}} constructs a comparator of type 'a comparator ⇒ 'b comparator ⇒ 'c comparator ⇒ 'z comparator ⇒ ('a, 'b, 'c, 'z) Comparator-Generator\text{.\textit{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.

\texttt{hide-type} type

\textbf{4.1 Lexicographic combination of order}

fun \text{\textit{comp-lex}} :: order list ⇒ order
where
\text{\textit{comp-lex}} (c # cs) = (case c of Eq ⇒ \text{\textit{comp-lex}} cs | - ⇒ c) |
\text{\textit{comp-lex}} [] = Eq

\textbf{4.2 Improved code for non-lazy languages}

The following equations will eliminate all occurrences of \text{\textit{comp-lex}} in the generated code of the comparators.

\texttt{lemma \textit{comp-lex-unfolds}:}
\textit{comp-lex} [] = Eq
\textit{comp-lex} [c] = c
\textit{comp-lex} (c # d # cs) = (case c of Eq ⇒ \textit{comp-lex} (d # cs) | Lt ⇒ Lt | Gt ⇒ Gt)
  \text{by (cases c, auto)+}

\textbf{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 \text{\textit{comp-lex}}.

\texttt{lemma \textit{comp-lex-eq}: \textit{comp-lex} os = Eq ⇐⇒ (\forall ord ∈ set os. ord = Eq)}
  \text{by (induct os) (auto split: order.splits)}

\texttt{definition \textit{trans-order :: order ⇒ order ⇒ order ⇒ bool where}}
\textit{trans-order} x y z ⇐⇒ x \neq Gt → y \neq Gt → z \neq Gt ∧ ((x = Lt ∨ y = Lt) → z = Lt)
lemma trans-orderI:
\[(x \neq Gt \implies y \neq Gt \implies z \neq Gt \wedge ((x = Lt \lor y = Lt) \implies z = Lt)) \implies\]
\text{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:
\[(\forall i < Suc x. P i) \iff P 0 \land (\forall i < x. P (Suc i))\]
using less-Suc-eq-0-disj by force

lemma comp-lex-trans:
assumes length \( xs = length ys \)
and length \( ys = length zs \)
and \( \forall i < length zs. \text{trans-order } (xs ! i) (ys ! i) (zs ! i) \)
shows \( \text{trans-order } (\text{comp-lex } xs) (\text{comp-lex } ys) (\text{comp-lex } zs) \)
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)
cases \( x \ y \ z \) rule: order.exhaust [case-product order.exhaust order.exhaust]
  auto simp: All-less-Suc dest: trans-orderD)
qed (simp add: trans-order-def)

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

declare comp-lex.simps [simp del]

definition peq-comp :: `'a comparator ⇒ 'a ⇒ bool
where
peq-comp \( acmp x \) \( \iff \) \( \forall y. acmp x y = Eq \iff x = y \)

lemma peq-compD: peq-comp \( acmp x \) \( \iff \) \( acmp x y = Eq \iff x = y \)
unfolding peq-comp-def by auto

lemma peq-compI: \( \forall y. acmp x y = Eq \iff x = y \) \( \implies \) peq-comp \( acmp x \)
unfolding peq-comp-def by auto

definition psym-comp :: `'a comparator ⇒ 'a ⇒ bool
where
psym-comp \( acmp x \) \( \iff \) \( \forall y. \text{invert-order } (acomp x y) = (acomp y x) \)
lemma psym-compD:
  assumes psym-comp acomp x
  shows invert-order (acomp x y) = (acomp y x)
  using assms unfolding psym-comp-def by blast

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

4.4 Separate properties of comparators

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

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

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

definition eq-comp :: 'a comparator \Rightarrow bool where
  eq-comp acomp \leftrightarrow (\forall x. peq-comp acomp x)

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

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

definition trans-comp :: 'a comparator \Rightarrow bool where
  trans-comp acomp \leftrightarrow (\forall x. ptrans-comp acomp x)

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

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

definition sym-comp :: 'a comparator \Rightarrow bool where
  sym-comp acomp \leftrightarrow (\forall x. 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-compD2].
{ assume acomp x y = Eq
with eq-compD[OF eq-compD2]
show x = y by blast
}
{ assume acomp x y = Lt and acomp y z = Lt
with trans-orderD[OF trans-compD[OF trans-compD], of x y z]
show acomp x z = Lt by auto
}
qed

lemma comparator-imp-eq-sym-trans:

17
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:
\[\text{length } [] = 0 \quad \text{length } (x \# xs) = \text{Suc } (\text{length } xs)\]
\[(x \# xs)!0 = x \quad (x \# xs)! (\text{Suc } n) = xs!n\] 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 \(\text{prod} :: (\text{compare}, \text{compare})\) compare.

- derive compare-order prod creates an instance \(\text{prod} :: (\text{compare}, \text{compare})\) compare-order.

- derive linorder prod creates an instance \(\text{prod} :: (\text{linorder}, \text{linorder})\) 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
\[\text{lt } x y = (\text{le } x y \land \neg \text{le } y x)\] (is ?a)
\[\text{le } x x\] (is ?b)
\[\text{le } x y \implies \text{le } y z \implies \text{le } x z\] (is ?c1 \implies ?c2 \implies ?c3)
\[\text{le } x y \implies \text{le } y x \implies x = y\] (is ?d1 \implies ?d2 \implies ?d3)
\[\text{le } x y \lor \text{le } y x\] (is ?e)

proof --

19
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
  ~/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 (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 \( x \ y \ z :: \text{bool} \)

show invert-order (comparator-bool \( x \ y \)) = comparator-bool \( y \ x \) by (cases \( x \), (cases \( y \), auto)+)

show comparator-bool \( x \ y \) = Eq \( \implies x = y \) by (cases \( x \), (cases \( y \), auto)+)

show comparator-bool \( x \ y \) = Lt \( \implies \) comparator-bool \( y \ z \) = Lt \( \implies \) comparator-bool \( x \ z \) = Lt

by (cases \( x \), (cases \( y \), auto), cases \( y \), (cases \( z \), auto)+)

qed

local-setup \( \langle \langle \text{Comparator-Generator}.	ext{register-foreign-comparator}@\{\text{typ unit}\} @\{\text{term comparator-unit}\} @\{\text{thm comparator-unit}\} \rangle \rangle \)

local-setup \( \langle \langle \text{Comparator-Generator}.	ext{register-foreign-comparator}@\{\text{typ bool}\} @\{\text{term comparator-bool}\} @\{\text{thm comparator-bool}\} \rangle \rangle \)

derive compare bool unit

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

lemma comparator-bool-comparator-of [compare-simps]:
\[ \text{comparator-bool} = \text{comparator-of} \]

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]:
\[ \text{comparator-unit} = \text{comparator-of} \]

proof (intro ext)

fix \( a \ b \)

show comparator-unit \( a \ b \) = comparator-of \( a \ b \)

unfolding comparator-of-def by auto

qed

derive \( \text{linorder} \) \( \text{compare-order bool unit} \)

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-Lezorder
  ~/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)
      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

  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)
  qed

22
show \(\texttt{?case}\)
  by (cases \(\texttt{ys}\), simp-all)
next
case (Cons \(\texttt{x}\) \(\texttt{xs}\) \(\texttt{ys}\)) note \(IH = \texttt{this}\)
thus \(\texttt{?case}\)
proof (cases \(\texttt{ys}\))
  case \(\texttt{Nil}\)
  thus \(\texttt{?thesis}\) by auto
next
case (Cons \(\texttt{y}\) \(\texttt{ys}\))
show \(\texttt{?thesis}\) unfolding \(\texttt{Cons}\)
  using \(IH[\texttt{of ys}]\)
  by (cases \(\texttt{x}\) \(\texttt{y}\) rule: linorder-cases, auto)
qed
qed
qed

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

instance \(\texttt{option} \colon\) (compare-order) compare-order
proof
  note [simp] = le-of-comp-def lt-of-comp-def comparator-of-def
  show \(\texttt{le-of-comp (compare :: 'a option comparator)} = \texttt{op \le}\)
    unfolding compare-option-def compare-is-comparator-of
  proof (intro ext)
    fix \(\texttt{xy1 xy2 :: 'a option}\)
    show \(\texttt{le-of-comp (comparator-option comparator-of \texttt{comparator-of}) xy1 xy2 = (xy1 \le xy2)}\)
      by (cases \(\texttt{xy1}\), (cases \(\texttt{xy2}\), auto split: if-splits))
  qed
show lt-of-comp (compare :: 'a option comparator) = op <
unfolding compare-option-def compare-is-comparator-of
proof (intro ext)
fix xy1 xy2 :: 'a option
show lt-of-comp (comparator-option comparator-of) xy1 xy2 = (xy1 < xy2)
by (cases xy1, (cases xy2, auto split: if-splits)+)
qed
qed
end

5 Checking Equality Without ”=”

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

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

In the following, we define a generator which for a given datatype ('a, 'b, 'c, 'z) Equality-Generator.type constructs an equality-test function of type ('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('c ⇒ 'c ⇒ bool) ⇒ ('z ⇒ 'z ⇒ bool) ⇒ ('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 equal-class must not be enforced.

hide-type type

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

definition list-all-eq :: bool list ⇒ bool where
list-all-eq = list-all id

5.1 Improved Code for Non-Lazy Languages

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

lemma list-all-eq-unfold:
list-all-eq [] = True
list-all-eq [b] = b
list-all-eq (b1 # b2 # bs) = (b1 ∧ list-all-eq (b2 # bs))
unfolding list-all-eq-def
by auto

end
lemma list-all-eq: list-all-eq bs ←→ (∀ b ∈ 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 ⇒ ‘a ⇒ bool

definition pequality :: ‘a equality ⇒ ‘a ⇒ bool
where
pequality aeq x ←→ (∀ y. aeq x y ←→ x = y)

lemma pequalityD: pequality aeq x ⇒ aeq x y ←→ x = y
unfolding pequality-def by auto

lemma pequalityI: (∀ x y. aeq x y ←→ x = y) ⇒ pequality aeq x
unfolding pequality-def by auto

5.3 Global equality property

definition equality :: ‘a equality ⇒ bool where
equality aeq ←→ (∀ x. pequality aeq x)

lemma equalityD2: equality aeq ⇒ pequality aeq x
unfolding equality-def by blast

lemma equalityI2: (∀ x. pequality aeq x) ⇒ equality aeq
unfolding equality-def by blast

lemma equalityD: equality aeq ⇒ aeq x y ←→ x = y
by (rule pequalityD[OF equalityD2])

lemma equalityI: (∀ x y. aeq x y ←→ x = y) ⇒ equality aeq
by (intro equalityI2 pequalityI)

lemma equality-imp-eq:
equality aeq ⇒ 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. 
\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{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 ⇒ 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 \texttt{hash\-combine} originates from evaluating the hash-
function on the arguments of a constructor, and the second argument of 
\texttt{hash\-combine} will be static \texttt{magic} numbers which are generated within the 
generator.

6.1 Improved Code for Non-Lazy Languages

lemma hash-combine-unfold:
| hash-combine [] [x] = x
| hash-combine (y # ys) (z # zs) = y * z + hash-combine ys zs
by auto
6.2 The Generator
ML-file hash-generator.ML

end

6.3 Defining Hash-Functions for Common Types

theory Hash-Instances
imports Hash-Generator
begin

For all of the following types, we register hashcode-functions. int, integer, nibble, 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 nibble bool char unit nat

derive hash-code prod sum option list

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

end

7 Countable Datatypes

theory Countable-Generator
imports ~~/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 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

27
val - = writeln (proving that datatype `base-name` is countable)
val sort = @{sort countable}
val vs =
  let val i = BNF-LFP-Compat.the-spec thy dtyp-name |> #1
  in map (fn (n,-) => (n, sort)) i end
val thy' = Class.instantiation ([dtyp-name],vs,sort) thy
  |> Class.prove-instantiation-exit (fn ctxt => countable-tac ctxt 1)
val - = writeln (registered `base-name` in class countable)
  in thy' end
in
  Derive-Manager.register-derive countable register datatypes is class countable
derive
end

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 \texttt{compare-order}-instance.
\begin{verbatim}
derive (linorder) compare-order rat
  Use \texttt{op} = as equality function.
derive (eq) equality rat
  First manually define a hashcode function.
\end{verbatim}

\begin{verbatim}
instantiation rat :: hashable
begin
definition def-hashmap-size = (\lambda :: rat itself. 10)
definition hashcode (r :: rat) = hashcode (quotient-of r)
instance
  by (intro-classes)(simp-all add: def-hashmap-size-rat-def)
end
And then register it at the generator.
derive (hashcode) hash-code rat
\end{verbatim}

9.2 A Datatype Without Nested Recursion
\begin{verbatim}
datatype 'a bintree = BEmpty | BNode 'a bintree 'a 'a bintree
\end{verbatim}
\begin{verbatim}
derive compare-order bintree
derive countable bintree
derive equality bintree
derive hashable bintree
\end{verbatim}

9.3 Using Other datatypes
\begin{verbatim}
datatype nat-list-list = NNil | CCons nat list × rat option nat-list-list
\end{verbatim}
\begin{verbatim}
derive compare-order nat-list-list
derive countable nat-list-list
derive (eq) equality nat-list-list
derive hashable nat-list-list
\end{verbatim}

9.4 Mutual Recursion
\begin{verbatim}
datatype 'a mtree = MEmpty | MNode 'a 'a mtree-list and
  'a mtree-list = MNil | MCons 'a mtree 'a mtree-list
\end{verbatim}
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 \textit{complex2} we only derive \textit{compare} which is not a subclass of \textit{linorder}. The instance will be \textit{complex2} :: (\textit{compare}, \textit{compare}) \textit{compare}, i.e., again the argument types have to be in class \textit{compare}.

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

\textbf{10 Acknowledgements}

We thank

\begin{itemize}
  \item Lukas Bulwahn and Brian Huffman for the discussion on a generic \textit{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

