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 project† [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 ............................................ 6
   3.3 Example: Modifying the Code-Equations of Red-Black-Trees 7
   3.4 A Comparator-Interface to Red-Black-Trees .......... 8

---

*Supported by FWF (Austrian Science Fund) projects P27502 and Y757.
†http://cl-informatik.uibk.ac.at/software/ceta
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4.1</td>
<td>Lexicographic combination of order</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>4.2</td>
<td>Improved code for non-lazy languages</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>4.3</td>
<td>Pointwise properties for equality, symmetry, and transitivity</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>4.4</td>
<td>Separate properties of comparators</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>4.5</td>
<td>Auxiliary Lemmas for Comparator Generator</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4.6</td>
<td>The Comparator Generator</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4.7</td>
<td>Compare Generator</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4.8</td>
<td>Defining Comparators and Compare-Instances for Common Types</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>4.9</td>
<td>Defining Compare-Order-Instances for Common Types</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>5.1</td>
<td>Improved Code for Non-Lazy Languages</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>Partial Equality Property</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>5.3</td>
<td>Global equality property</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>5.4</td>
<td>The Generator</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>5.5</td>
<td>Defining Equality-Functions for Common Types</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>6.1</td>
<td>Improved Code for Non-Lazy Languages</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>The Generator</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>6.3</td>
<td>Defining Hash-Functions for Common Types</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>7.1</td>
<td>Installing the tactic</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>Loading Existing Derive-Commands</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>9.1</td>
<td>Rational Numbers</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>9.2</td>
<td>A Datatype Without Nested Recursion</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>9.3</td>
<td>Using Other datatypes</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>9.4</td>
<td>Mutual Recursion</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>9.5</td>
<td>Nested recursion</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>9.6</td>
<td>Examples from IsaFoR</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>9.7</td>
<td>A Complex Datatype</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>Acknowledgements</td>
<td>26</td>
</tr>
</tbody>
</table>

1 Derive Manager

```plaintext
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.

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

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

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

\texttt{(ML)}

end

2 Shared Utilities for all Generator

In this theory we mainly provide some Isabelle/ML infrastructure that is used by several generators. It consists of a uniform interface to access all the theorems, terms, etc. from the BNF package, and some auxiliary functions which provide recursors on datatypes, common tactics, etc.

theory Generator-Aux

imports Main

begin

\texttt{(ML)}

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

(proof)

lemma \texttt{conj-weak-cong:} \ a = b \implies c = d \implies (a \land c) = (b \land d) \texttt{(proof)}
lemma refl-True: \( (x = x) = True \) (proof)

end

3 Comparisons

3.1 Comparators and Linear Orders

theory Comparator
imports Main

begin

Instead of having to define a strict and a weak linear order, \((op <, op \leq)\), 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 ⇒ 'a ⇒ order

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

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

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

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

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

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

lemma le-of-comp-of-ords-gen: (∀ x y. lt x y ⇒ le x y) ⇒ le-of-comp (comp-of-ords le lt) = le (proof)

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

fun invert-order:: order ⇒ order where
invert-order Lt = Gt |
invert-order Gt = Lt |


invert-order $Eq = Eq$

locale comparator =
  fixes comp :: \textquoteleft a comparator
  assumes sym: invert-order $(\text{comp } x \ y) = \text{comp } y \ x$
  and weak-eq: $\text{comp } x \ y = Eq \implies x = y$
  and trans: $\text{comp } x \ y = Lt \implies \text{comp } y \ z = Lt \implies \text{comp } x \ z = Lt$

begin

lemma eq: $(\text{comp } x \ y = Eq) = (x = y)$$\langle proof \rangle$

lemma comp-same [simp]:
  $\text{comp } x \ x = Eq$$\langle proof \rangle$

abbreviation lt $\equiv \text{lt-of-comp } \text{comp}$
abbreviation le $\equiv \text{le-of-comp } \text{comp}$

lemma linorder: class.linorder le lt
$\langle proof \rangle$

sublocale linorder le lt
$\langle proof \rangle$

lemma Gt-lt-conv: $\text{comp } x \ y = Gt \iff \text{lt } y \ x$$\langle proof \rangle$

lemma Lt-lt-conv: $\text{comp } x \ y = Lt \iff \text{lt } x \ y$$\langle proof \rangle$

lemma eq-Eq-conv: $\text{comp } x \ y = Eq \iff x = y$$\langle proof \rangle$

lemma nGt-le-conv: $\text{comp } x \ y \neq Gt \iff \text{le } x \ y$$\langle proof \rangle$

lemma nLt-le-conv: $\text{comp } x \ y \neq Lt \iff \text{le } y \ x$$\langle proof \rangle$

lemma nEq-neq-conv: $\text{comp } x \ y \neq Eq \iff x \neq y$$\langle proof \rangle$

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:
  $(\text{if } le \ x \ y \text{ then } (\text{if } x = y \text{ then } P \text{ else } Q) \text{ else } R) = (\text{case-order } P \ Q \ R \ (\text{comp } x \ y))$
  $(\text{if } le \ x \ y \text{ then } (\text{if } y = x \text{ then } P \text{ else } Q) \text{ else } R) = (\text{case-order } P \ Q \ R \ (\text{comp } x \ y))$
  $(\text{if } le \ x \ y \text{ then } (\text{if } le \ y \ x \text{ then } P \text{ else } Q) \text{ else } R) = (\text{case-order } P \ Q \ R \ (\text{comp } x \ y))$
  $(\text{if } le \ x \ y \text{ then } (\text{if } le \ x \ y \text{ then } Q \text{ else } P) \text{ else } R) = (\text{case-order } P \ Q \ R \ (\text{comp } x \ y))$
  $(\text{if } lt \ x \ y \text{ then } Q \text{ else } (\text{if } le \ x \ y \text{ then } P \text{ else } R)) = (\text{case-order } P \ Q \ R \ (\text{comp } x \ y))$
  $(\text{if } lt \ x \ y \text{ then } Q \text{ else } (\text{if } lt \ y \ x \text{ then } R \text{ else } P)) = (\text{case-order } P \ Q \ R \ (\text{comp } x \ y))$
  $(\text{if } lt \ x \ y \text{ then } Q \text{ else } (\text{if } x = y \text{ then } P \text{ else } R)) = (\text{case-order } P \ Q \ R \ (\text{comp } x \ y))$
(if \lt x y then Q else (if y = x then P else R)) = (case-order P Q R (comp x y))
(if x = y then P else (if \lt y x then R else Q)) = (case-order P Q R (comp x y))
(if x = y then P else (if \lt x y then Q else R)) = (case-order P Q R (comp x y))
(if x = y then P else (if le y x then R else Q)) = (case-order P Q R (comp x y))
(if x = y then P else (if le x y then Q else R)) = (case-order P Q R (comp x y))

⟨proof⟩
end

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

definition (in linorder) comparator-of :: 'a comparator where
  comparator-of x y = (if x < y then Lt else if x = y then Eq else Gt)

lemma comparator-of: comparator comparator-of
⟨proof⟩
end

3.2 Compare

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

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

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

lemma compare-Eq-is-eq [simp]:
  compare x y = Eq \longleftrightarrow x = y
⟨proof⟩

lemma compare-refl [simp]:
  compare x x = Eq
⟨proof⟩

end
lemma (in linorder) le-lt-comparator-of:
le-of-comp comparator-of = op ≤ lt-of-comp comparator-of = op <
⟨proof⟩

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
⟨proof⟩

class context compare-order begin

lemma compare-is-comparator-of:
compare = comparator-of
⟨proof⟩

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

thm two-comparisons-into-compare
end
⟨ML⟩

Compare-Code.change-compare-code const ty−vars changes the code equations of some constant such that two consecutive comparisons via op ≤, op <", or op = are turned into one invocation of compare. The difference to a standard code-unfold is that here we change the code-equations where an additional sort-constraint on compare-order can be added. Otherwise, there would be no compare-function.
end

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

theory RBT-Compare-Order-Impl
imports
  Compare
  ~~/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.

\begin{verbatim}
compare-code ('a) rbt-ins
compare-code ('a) rbt-lookup
compare-code ('a) rbt-del
compare-code ('a) rbt-map-entry
compare-code ('a) union-with
compare-code ('a) inter-with

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

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
where
  rbt-comp-ins f k v RBT-Impl.Empty = Branch RBT-Impl.R RBT-Impl.Empty k v
  rbt-comp-ins f k v (Branch RBT-Impl.B l x y r) = (case c k x of
    Lt ⇒ balance (rbt-comp-ins f k v l) x y r
  | Gt ⇒ balance l x y (rbt-comp-ins f k v r)

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

\textbf{definition} \textit{rbt-comp-insert} :: 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt
\textbf{where} \textit{rbt-comp-insert} = \textit{rbt-comp-insert-with-key} (\lambda - \textit{nu}, \textit{nv})

\textbf{fun}
\textit{rbt-comp-del-from-left} :: 'a ⇒ ('a,'b) rbt ⇒ ('a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt
\textbf{and}
\textit{rbt-comp-del-from-right} :: 'a ⇒ ('a,'b) rbt ⇒ ('a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt
\textbf{and}
\textit{rbt-comp-del} :: 'a⇒ ('a,'b) rbt ⇒ ('a,'b) rbt
\textbf{where}
\textit{rbt-comp-del x RBT-Impl.Empty} = \textit{RBt-Impl.Empty} | 
\textit{rbt-comp-del x} (Branch - a y s b) = 
\text{(case c x y of} 
\text{Lt} ⇒ \textit{rbt-comp-del-from-left} x a y s b 
| \text{Gt} ⇒ \textit{rbt-comp-del-from-right} x a y s b 
| \text{Eq} ⇒ \textit{combine a b} | 
\textit{rbt-comp-del-from-left} x (Branch RBT-Impl.B lt z v rt) y s b = \textit{balance-left} 
(\textit{rbt-comp-del x} (Branch RBT-Impl.B lt z v rt)) y s b | 
\textit{rbt-comp-del-from-left} x a y s b = \textit{Branch RBT-Impl.R} (\textit{rbt-comp-del x} a) y s b | 
\textit{rbt-comp-del-from-right} x a y s (Branch RBT-Impl.B lt z v rt) = \textit{balance-right} a y s (\textit{rbt-comp-del x} (Branch RBT-Impl.B lt z v rt)) | 
\textit{rbt-comp-del-from-right} x a y s b = \textit{Branch RBT-Impl.R} a y s (\textit{rbt-comp-del x} b)

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

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

\textbf{primrec}
\textit{rbt-comp-map-entry} :: 'a ⇒ ('b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
\textbf{where}
\textit{rbt-comp-map-entry} k f RBT-Impl.Empty = RBT-Impl.Empty
\textbf{|} \textit{rbt-comp-map-entry} k f (Branch cc lt x v rt) = 
\text{(case c k x of} 
\text{Lt} ⇒ Branch cc (\textit{rbt-comp-map-entry} k f lt) x v rt 
| \text{Gt} ⇒ Branch cc lt x v (\textit{rbt-comp-map-entry} k f rt) 
| \text{Eq} ⇒ Branch cc lt x (f v) rt) 

\textbf{function} \textit{comp-sunion-with} :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a × 'b) list
\textit{⇒ ('a × 'b) list}

9
where
\[ \text{comp-sunion-with } f \ ((k, v) \# as) ((k', v') \# bs) = \]
\[(\text{case } c k k'\text{ of} \]
\[ \text{Lt } \Rightarrow (k', v') \# \text{comp-sunion-with } f ((k, v) \# as) bs \]
\[ \text{Gt } \Rightarrow (k, v) \# \text{comp-sunion-with } f as ((k', v') \# bs) \]
\[ \text{Eq } \Rightarrow (k, f k v v') \# \text{comp-sunion-with } f as bs \]
\[ \text{comp-sunion-with } f [] bs = bs \]
\[ \text{comp-sunion-with } f as [] = as \]
\end{align*}
\text{(proof)}

termination (\text{proof})

\begin{align*}
\text{function } \text{comp-sinter-with } :: (a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow (\text{'a } \times \text{'b}) \text{list } \Rightarrow (\text{'a } \times \text{'b}) \text{list} \\
\text{where}
\[ \text{comp-sinter-with } f ((k, v) \# as) ((k', v') \# bs) = \]
\[(\text{case } c k k'\text{ of} \]
\[ \text{Lt } \Rightarrow \text{comp-sinter-with } f ((k, v) \# as) bs \]
\[ \text{Gt } \Rightarrow \text{comp-sinter-with } f as ((k', v') \# bs) \]
\[ \text{Eq } \Rightarrow (k, f k v v') \# \text{comp-sinter-with } f as bs \]
\[ \text{comp-sinter-with } f [] - = [] \]
\[ \text{comp-sinter-with } f - [] = [] \]
\end{align*}
\text{(proof)}

termination (\text{proof})

\begin{align*}
\text{definition } \text{rbt-comp-union-with-key } :: (a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow (\text{'a } \times \text{'b}) \text{rbt } \Rightarrow (\text{'a }, 'b) \text{rbt } \Rightarrow (\text{'a }, 'b) \text{rbt} \\
\text{where}
\[ \text{rbt-comp-union-with-key } f t1 t2 = \]
\[(\text{case } \text{RBT-Impl.compare-height } t1 t1 t2 t2 \]
\[ \text{of } \text{compare.EQ } \Rightarrow \text{rbtreeify } (\text{comp-sunion-with } f (\text{RBT-Impl.entries } t1)) (\text{RBT-Impl.entries } t2)) \]
\[ \text{compare.LT } \Rightarrow \text{RBT-Impl.fold } (\text{rbt-comp-insert-with-key } (\lambda k v w. f k w v)) t1 t2 \]
\[ \text{compare.GT } \Rightarrow \text{RBT-Impl.fold } (\text{rbt-comp-insert-with-key } f) t2 t1 \]
\end{align*}

\text{definition } \text{rbt-comp-inter-with-key } :: (a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow (\text{'a } \times \text{'b}) \text{rbt } \Rightarrow (\text{'a }, 'b) \text{rbt} \\
\text{where}
\[ \text{rbt-comp-inter-with-key } f t1 t2 = \]
\[(\text{case } \text{RBT-Impl.compare-height } t1 t1 t2 t2 \]
\[ \text{of } \text{compare.EQ } \Rightarrow \text{rbtreeify } (\text{comp-sinter-with } f (\text{RBT-Impl.entries } t1)) (\text{RBT-Impl.entries } t2)) \]
\[ \text{compare.LT } \Rightarrow \text{rbtreeify } (\text{List.map-filter } (\lambda (k, v). \text{map-option } (\lambda w. (k, f k v w))) (\text{rbt-comp-lookup } t1 t2 k)) (\text{RBT-Impl.entries } t1)) \]
\[ \text{compare.GT } \Rightarrow \text{rbtreeify } (\text{List.map-filter } (\lambda (k, v). \text{map-option } (\lambda w. (k, f k w v))) (\text{rbt-comp-lookup } t1 t2 k)) (\text{RBT-Impl.entries } t2)) \]
\end{align*}

context
assumes \( c : \text{comparator } c \)

begin

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

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

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

lemma \( \text{rbt-comp-insert} : \text{rbt-comp-insert} = \text{ord.rbt-insert} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemma \( \text{rbt-comp-del} : \text{rbt-comp-del} = \text{ord.rbt-del} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemma \( \text{rbt-comp-delete} : \text{rbt-comp-delete} = \text{ord.rbt-delete} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemma \( \text{rbt-comp-bulkload} : \text{rbt-comp-bulkload} = \text{ord.rbt-bulkload} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemma \( \text{rbt-comp-map-entry} : \text{rbt-comp-map-entry} = \text{ord.rbt-map-entry} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemma \( \text{comp-sunion-with} : \text{comp-sunion-with} = \text{ord.sunion-with} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemma \( \text{comp-sinter-with} : \text{comp-sinter-with} = \text{ord.sinter-with} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemma \( \text{rbt-comp-union-with-key} : \text{rbt-comp-union-with-key} = \text{ord.rbt-union-with-key} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemma \( \text{rbt-comp-inter-with-key} : \text{rbt-comp-inter-with-key} = \text{ord.rbt-inter-with-key} \ (\text{lt-of-comp } c) \) 
\langle proof \rangle

lemmas \( \text{rbt-comp-simps} = \
\text{rbt-comp-insert} \
\text{rbt-comp-lookup} \
\text{rbt-comp-delete} \
\text{rbt-comp-bulkload} \
\text{rbt-comp-map-entry} \)
4 Generating Comparators

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

typedec ('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) | Lt ⇒ Lt | Gt ⇒ Gt)
⟨proof⟩
4.3 Pointwise properties for equality, symmetry, and transitivity

The pointwise properties are important during inductive proofs of soundness of comparators. They are defined in a way that are combinable with comp-lex.

**Lemma** `comp-lex-eq`: `comp-lex os = Eqifetime (∀ ord ∈ set os. ord = Eq)

**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 ≠ Gt ⇒ y ≠ Gt ⇒ z ≠ Gt ∧ ((x = Lt ∨ y = Lt) → z = Lt)) ⇒ trans-order x y z
```

**Lemma** `trans-orderD`:

```
assumes trans-order x y z and x ≠ Gt and y ≠ Gt
shows z ≠ Gt and x = Lt ∨ y = Lt ⇒ z = Lt
```

**Lemma** `All-less-Suc`:

```
(∀ i < Suc x. P i) ←→ P 0 ∧ (∀ i < x. P (Suc i))
```

**Lemma** `comp-lex-trans`:

```
assumes length xs = length ys
and length ys = length zs
and ∀ i < length zs. trans-order (xs ! i) (ys ! i) (zs ! i)
shows trans-order (comp-lex xs) (comp-lex ys) (comp-lex zs)
```

**Lemma** `comp-lex-sym`:

```
assumes length xs = length ys
and ∀ i < length ys. invert-order (xs ! i) = ys ! i
shows invert-order (comp-lex xs) = comp-lex ys
```

**Declare** `comp-lex.simps [simp del]`

**Definition** `peq-comp :: 'a comparator ⇒ 'a ⇒ bool` where

```
peq-comp acomp x ←→ (∀ y. acomp x y = Eq ←→ x = y)
```

**Lemma** `peq-compD`: `peq-comp acomp x ⇒ acomp x y = Eq ←→ x = y

(proof)```
\textbf{lemma} \textit{peq-compI}: \((\forall y. \text{acomp } x y = \text{Eq} \iff x = y) \implies \text{peq-comp } \text{acomp } x\) \\
\langle proof \rangle 

\textbf{definition} \textit{psym-comp} :: \('a\text{ comparator } \Rightarrow 'a \Rightarrow \text{bool}\) where \\
\textit{psym-comp } \text{acomp } x \iff (\forall y. \text{invert-order } (\text{acomp } x y) = (\text{acomp } y x))

\textbf{lemma} \textit{psym-compD}: \\
\textbf{assumes} \textit{psym-comp } \text{acomp } x \\
\textbf{shows} \text{invert-order } (\text{acomp } x y) = (\text{acomp } y x) \\
\langle proof \rangle 

\textbf{lemma} \textit{psym-compI}: \\
\textbf{assumes} \(\forall y. \text{invert-order } (\text{acomp } x y) = (\text{acomp } y x)\) \\
\textbf{shows} \textit{psym-comp } \text{acomp } x \\
\langle proof \rangle 

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

\textbf{lemma} \textit{ptrans-compD}: \\
\textbf{assumes} \textit{ptrans-comp } \text{acomp } x \\
\textbf{shows} \text{trans-order } (\text{acomp } x y) (\text{acomp } y z) (\text{acomp } x z) \\
\langle proof \rangle 

\textbf{lemma} \textit{ptrans-compI}: \\
\textbf{assumes} \(\forall y z. \text{trans-order } (\text{acomp } x y) (\text{acomp } y z) (\text{acomp } x z)\) \\
\textbf{shows} \textit{ptrans-comp } \text{acomp } x \\
\langle proof \rangle 

4.4 Separate properties of comparators

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

\textbf{lemma} \textit{eq-compD2}: \textit{eq-comp } \text{acomp } \implies \text{peq-comp } \text{acomp } x \\
\langle proof \rangle 

\textbf{lemma} \textit{eq-compI2}: \((\forall x. \text{peq-comp } \text{acomp } x) \implies \textit{eq-comp } \text{acomp}\) \\
\langle proof \rangle 

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

\textbf{lemma} \textit{trans-compD2}: \textit{trans-comp } \text{acomp } \implies \text{ptrans-comp } \text{acomp } x \\
\langle proof \rangle 

\textbf{lemma} \textit{trans-compI2}: \((\forall x. \text{ptrans-comp } \text{acomp } x) \implies \textit{trans-comp } \text{acomp}\)
definition sym-comp :: 'a comparator ⇒ bool where
sym-comp acomp ⟷ (∀ x. psym-comp acomp x)

lemma sym-compD2:
sym-comp acomp ⟷ psym-comp acomp x
  ⟨proof⟩

lemma sym-compI2: (∀ x. psym-comp acomp x) ⇒ sym-comp acomp
  ⟨proof⟩

lemma eq-compD: eq-comp acomp ⟷ acomp x y = Eq ⟷ x = y
  ⟨proof⟩

lemma eq-compI: (∀ x y. acomp x y = Eq ⟷ x = y) ⇒ eq-comp acomp
  ⟨proof⟩

lemma trans-compD: trans-comp acomp ⇒ trans-order (acomp x y) (acomp y z)
  (acomp x z)
  ⟨proof⟩

lemma trans-compI: (∀ x y z. trans-order (acomp x y) (acomp y z) (acomp x z))
  ⇒ trans-comp acomp
  ⟨proof⟩

lemma sym-compD:
sym-comp acomp ⇒ invert-order (acomp x y) = (acomp y x)
  ⟨proof⟩

lemma sym-compI: (∀ x y. invert-order (acomp x y) = (acomp y x)) ⇒ sym-comp acomp
  ⟨proof⟩

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

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

context
  fixes acomp :: 'a comparator
  assumes c: comparator acomp
begin
lemma comp-to-psym-comp: psym-comp acomp x
⟨proof⟩

lemma comp-to-peq-comp: peq-comp acomp x
⟨proof⟩

lemma comp-to-ptrans-comp: ptrans-comp acomp x
⟨proof⟩

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)))
⟨proof⟩

lemma trans-order-different:
trans-order a b Lt
trans-order Gt b c
trans-order a Gt c
⟨proof⟩

lemma length-nth-simps:
length [] = 0 length (x # xs) = Suc (length xs)
(x # xs) ! 0 = x (x # xs) ! (Suc n) = xs ! n ⟨proof⟩

4.6 The Comparator Generator
⟨ML⟩

end

4.7 Compare Generator

theory Compare-Generator
imports
  Comparator-Generator
  Compare
begin

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

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

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

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

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

lemma linorder-axiomsD: assumes class.linorder le lt
shows
  lt x y = (le x y ∧ ¬ 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⟩

named-theorems compare-simps simp theorems to derive compare = comparator-of
⟨ML⟩
end

4.8 Defining Comparators and Compare-Instances for Common Types

theory Compare-Instances
imports
  Compare-Generator
  ~/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
   ⟨proof⟩

lemma comparator-bool: comparator comparator-bool
   ⟨proof⟩

⟨ML⟩

derive compare bool unit

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

lemma comparator-bool-comparator-of [compare-simps]:
   comparator-bool = comparator-of
   ⟨proof⟩

lemma comparator-unit-comparator-of [compare-simps]:
   comparator-unit = comparator-of
   ⟨proof⟩

derive (linorder) compare-order bool unit
end

4.9 Defining Compare-Order-Instances for Common Types

theory Compare-Order-Instances
imports
   Compare-Instances
   ~/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`
langle proof \rangle

instance `prod :: (compare-order, compare-order)compare-order`
langle proof \rangle

instance `option :: (compare-order)compare-order`
langle proof \rangle

declare

typedecl \langle \text{′}_a, \text{′}_b, \text{′}_c, \text{′}_z \rangle\text{type}

declaration

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) \text{Equality-Generator.type} ⇒ (′_a, ′_b, ′_c, ′_z) \text{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))
 ⟨proof⟩

lemma list-all-eq: list-all-eq bs ⇔ (∀ b ∈ set bs. b)
 ⟨proof⟩

5.2 Partial Equality Property

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

type-synonym 'a equality = 'a ⇒ '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
 ⟨proof⟩

lemma pequalityI: (∀ x y. aeq x y ⇔ x = y) ⇒ pequality aeq x
 ⟨proof⟩

5.3 Global equality property

definition equality :: 'a equality ⇒ bool where
  equality aeq ⇔ (∀ x. pequality aeq x)

lemma equalityD2: equality aeq ⇒ pequality aeq x
 ⟨proof⟩

lemma equalityI2: (∀ x. pequality aeq x) ⇒ equality aeq
 ⟨proof⟩

lemma equalityD: equality aeq ⇒ aeq x y ⇔ x = y
 ⟨proof⟩

lemma equalityI: (∀ x y. aeq x y ⇔ x = y) ⇒ equality aeq
 ⟨proof⟩

lemma equality-imp-eq:
  equality aeq ⇒ aeq = (op =)
 ⟨proof⟩

lemma eq-equality: equality (op =)
 ⟨proof⟩

lemma equality-def': equality f = (f = op =)
5.4 The Generator

\[\text{hide-fact (open) equalityI}\]
end

5.5 Defining Equality-Functions for Common Types

theory Equality-Instances
imports Equality-Generator
begin

For all of the following types, we register equality-functions. \(\text{int, integer, nibble, nat, char, bool, unit, sum, option, list, and prod}\). For types without type parameters, we use plain \(op =\), 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
\begin{align*}
\text{hash-combine } [] & \mid x = x \\
\text{hash-combine } (y \# ys) & \mid (z \# zs) = y \ast z + \text{hash-combine } ys zs \\
\text{hash-combine } - & = 0
\end{align*}

The first argument of \text{hash-combine} originates from evaluating the hash-function on the arguments of a constructor, and the second argument of \text{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 [] [x] = x
  hash-combine (y # ys) (z # zs) = y * z + hash-combine ys zs
⟨proof⟩

6.2 The Generator
⟨ML⟩
end

6.3 Defining Hash-Functions for Common Types

theory Hash-Instances
imports
  Hash-Generator
begin

  For all of the following types, we register hashcode-functions. int, integer, 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.

⟨ML⟩
end

8 Loading Existing Derive-Commands

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

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

  print-derives

end

9 Examples

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

9.1 Rational Numbers

The rational numbers are not a datatype, so it will not be possible to derive corresponding instances of comparators, hashcodes, etc. via the generators. But we can and should still register the existing instances, so that later datatypes are supported which use rational numbers.

  Use the linear order on rationals to define the compare-order-instance.

derive (linorder) compare-order rat

  Use op = as equality function.
derive \( (eq) \) equality rat

First manually define a hashcode function.

**instantiation** rat :: hashable

**begin**

**definition** def-hashmap-size = \((\lambda - :: \text{rat itself}.\ 10)\)

**definition** hashcode \((r :: \text{rat}) = \text{hashcode (quotient-of r)}\)

**instance**

\((\text{proof})\)

**end**

And then register it at the generator.

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

9.2 A Datatype Without Nested Recursion

**datatype** 'a bintree = BEmpty | BNode 'a bintree 'a 'a bintree

**derive** compare-order bintree

**derive** countable bintree

**derive** equality bintree

**derive** hashable bintree

9.3 Using Other datatypes

**datatype** nat-list-list = NNil | CCons nat list × rat option nat-list-list

**derive** compare-order nat-list-list

**derive** countable nat-list-list

**derive** \((eq) \text{ equality nat-list-list}\)

**derive** hashable nat-list-list

9.4 Mutual Recursion

**datatype**

\( 'a \text{ mtree} = MEmpty | MNode 'a 'a \text{ mtree-list and} \)

\( 'a \text{ mtree-list} = MNil | MCons 'a \text{ mtree} 'a \text{ mtree-list} \)

**derive** compare-order mtree mtree-list

**derive** countable mtree mtree-list

**derive** hashable mtree mtree-list

For **derive** \((equality|comparator|hash-code) \text{ mutual-recursive-type}\) 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 \text{mtree} and \text{mtree-list} suffices.

**derive** equality mtree

9.5 Nested recursion

**datatype** 'a tree = Empty | Node 'a 'a tree list
9.6 Examples from IsaFoR

datatype $(f,v)$ term = $\text{Var } v \mid \text{Fun } f (f,v)$ term list
datatype $(f, l)$ lab =
   Lab $(f, l)$ lab $l$
   | FunLab $(f, l)$ lab $(f, l)$ lab list
   | UnLab $f$
   | Sharp $(f, l)$ lab
derive compare-order term lab
derive countable term lab
derive equality term lab
derive hashable term lab

9.7 A Complex Datatype

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

datatype $(a, b)$ complex =
   $C1$ nat $a$ ttree $\times$ rat $+$ $(a,b)$ complex list $|
   C2$ $(a,b)$ complex list ttree $b$ $(a,b)$ complex $(a,b)$ complex2 ttree list
and $(a, b)$ complex2 = $D1$ $(a, b)$ complex ttree

On this last example type we illustrate the difference of the various comparator- and order-generators.

For complex we create an instance of compare-order which also defines a linear order. Note however that the instance will be complex :: (compare, compare) compare-order, i.e., the argument types have to be in class compare.

For complex2 we only derive compare which is not a subclass of linorder. The instance will be complex2 :: (compare, compare) compare, i.e., again the argument types have to be in class compare.

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

derive compare-order complex
derive compare complex2
derive linorder complex2
derive countable complex complex2
derive equality complex
derive hashable complex complex2

end

10 Acknowledgements

We thank

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

References

