Deriving class instances for datatypes.*

Christian Sternagel and René Thiemann

February 23, 2021

Abstract

We provide a framework for registering automatic methods to derive class instances of datatypes, as it is possible using Haskell’s “deriving Ord, Show, …” feature.

We further implemented such automatic methods to derive comparators, linear orders, parametrizable equality functions, and hash-functions which are required in the Isabelle Collection Framework [1] and the Container Framework [2]. Moreover, for the tactic of Blanchette to show that a datatype is countable, we implemented a wrapper so that this tactic becomes accessible in our framework. All of the generators are based on the infrastructure that is provided by the BNF-based datatype package.

Our formalization was performed as part of the IsaFoR/CeTA project[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 Shared Utilities for all Generator
3 Comparisons
   3.1 Comparators and Linear Orders
   3.2 Compare
   3.3 Example: Modifying the Code-Equations of Red-Black-Trees
   3.4 A Comparator-Interface to Red-Black-Trees

*Supported by FWF (Austrian Science Fund) projects P27502 and Y757.
4 Generating Comparators

4.1 Lexicographic combination of order .................................. 12
4.2 Improved code for non-lazy languages ................................. 12
4.3 Pointwise properties for equality, symmetry, and transitivity ....... 13
4.4 Separate properties of comparators ................................... 14
4.5 Auxiliary Lemmas for Comparator Generator ......................... 16
4.6 The Comparator Generator ............................................. 16
4.7 Compare Generator ..................................................... 16
4.8 Defining Comparators and Compare-Instances for Common Types .................................................. 17
4.9 Defining Compare-Order-Instances for Common Types .......... 18
4.10 Compare Instance for Rational Numbers ............................ 19
4.11 Compare Instance for Real Numbers ................................ 19

5 Checking Equality Without "="

5.1 Improved Code for Non-Lazy Languages ............................ 20
5.2 Partial Equality Property .............................................. 20
5.3 Global equality property .............................................. 21
5.4 The Generator .......................................................... 21
5.5 Defining Equality-Functions for Common Types .................... 21

6 Generating Hash-Functions

6.1 Improved Code for Non-Lazy Languages ............................ 22
6.2 The Generator .......................................................... 22
6.3 Defining Hash-Functions for Common Types ......................... 22

7 Countable Datatypes

7.1 Installing the tactic ..................................................... 23

8 Loading Existing Derive-Commands

9 Examples

9.1 Rational Numbers ...................................................... 24
9.2 A Datatype Without Nested Recursion .............................. 24
9.3 Using Other datatypes ................................................. 25
9.4 Mutual Recursion ...................................................... 25
9.5 Nested recursion ....................................................... 25
9.6 Examples from IsaFoR ............................................... 25
9.7 A Complex Datatype ................................................ 26

10 Acknowledgements

2
1 Derive Manager

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

The derive manager allows the user to register various derive-hooks, e.g., for orders, pretty-printers, hash-functions, etc. All registered hooks are accessible via the derive command.

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

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

There is also the diagnostic command print-derives that shows the list of currently registered hooks.
⟨ML⟩
end

2 Shared Utilities for all Generator

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

theory Generator-Aux
imports Main
begin
⟨ML⟩

lemma in-set-simps:
  x ∈ set (y # z # ys) = (x = y ∨ x ∈ set (z # ys))
  x ∈ set [y] = (x = y)
  x ∈ set [] = False
  Ball (set []) P = True
  Ball (set [x]) P = P x

3
Ball (set (x ≠ y ≠ zs)) P = (P x ∧ Ball (set (y ≠ zs)) P)  
\langle proof \rangle

lemma conj-weak-cong: a = b ⇒ c = d ⇒ (a ∧ c) = (b ∧ d) \langle proof \rangle

lemma refl-True: (x = x) = True \langle proof \rangle
end

3 Comparisons

3.1 Comparators and Linear Orders

theory Comparator
imports Main
begin

Instead of having to define a strict and a weak linear order, ((<), (≤)), one can alternative use a comparator to define the linear order, which may deliver three possible outcomes when comparing two values.

datatype order = Eq | Lt | Gt

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

In the following, we provide the obvious definitions how to switch 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 
\langle proof \rangle

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

lemma le-of-comp-of-ords-gen: (\A x y. lt x y ⇒ le x y) ⇒ le-of-comp (comp-of-ords le lt) = le 
\langle proof \rangle

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

lemma comp-same [simp]:
  comp x x = Eq
  ⟨proof⟩

abbreviation lt ≡ lt-of-comp comp
abbreviation le ≡ le-of-comp comp

lemma linorder: class.linorder le lt
  ⟨proof⟩

sublocale linorder le lt
  ⟨proof⟩

lemma Gt-lt-conv: comp x y = Gt ←→ lt y x
  ⟨proof⟩
lemma Lt-lt-conv: comp x y = Lt ←→ lt x y
  ⟨proof⟩
lemma eq-Eq-conv: comp x y = Eq ←→ x = y
  ⟨proof⟩
lemma nGt-le-conv: comp x y ≠ Gt ←→ le x y
  ⟨proof⟩
lemma nLt-le-conv: comp x y ≠ Lt ←→ le y x
  ⟨proof⟩
lemma nEq-neq-conv: comp x y ≠ Eq ←→ x ≠ y
  ⟨proof⟩

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))
\[(\text{if } \leq x y \text{ then } (\text{if } \lt x y \text{ then } Q \text{ else } P) \text{ else } R) = (\text{case-order } P Q R (\text{comp } x y))\]

\[(\text{if } \lt x y \text{ then } Q \text{ else } (\text{if } \leq x y \text{ then } P \text{ else } R) = (\text{case-order } P Q R (\text{comp } x y))\]

\[(\text{if } \leq x y \text{ then } Q \text{ else } (\text{if } \lt y x \text{ then } R \text{ else } P) = (\text{case-order } P Q R (\text{comp } x y))\]

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

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

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

\[
\langle \text{proof} \rangle
\end{proof}

\begin{lemma}
\textit{comp-of-ords: assumes class.linorder le lt}
\textit{shows comparator (comp-of-ords le lt)}
\langle \text{proof} \rangle
\end{lemma}

\begin{definition}(in linorder) comparator-of :: 'a comparator where
comparator-of \text{ x y = (if } \text{ x y then } \text{ Lt else if } \text{ x y then } \text{ Eq else Gt)\}
\end{definition}

\begin{lemma}
\textit{comparator-of: comparator comparator-of}
\langle \text{proof} \rangle
\end{lemma}

\end

\section{3.2 Compare}

\textit{theory} \textit{Compare}

\textit{imports} \textit{Comparator}

\textit{keywords} \textit{compare-code :: thy-decl}

\textit{begin}

This introduces a type class for having a proper comparator, similar to \textit{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.

\begin{class}
\textit{compare =}
\textit{fixes compare :: 'a comparator}
\textit{assumes comparator-compare: comparator compare}
\end{class}

\begin{lemma}
\textit{compare-Eq-is-eq [simp]:}
\textit{compare x y = Eq \leftrightarrow x = y}
\langle \text{proof} \rangle
\end{lemma}

\begin{lemma}
\textit{compare-refl [simp]:}
\end{lemma}
\[ \text{compare } x \ x = \text{Eq} \]

\textit{proof}

\textbf{end}

\textbf{lemma (in linorder) le-lt-comparator-of:}
\[ \text{le-of-comp comparator-of } = (\leq) \ \text{lt-of-comp comparator-of } = (<) \]
\textit{proof}

\textbf{class compare-order = ord + compare +}
\textbf{assumes ord-defs: le-of-comp compare } = (\leq) \text{ lt-of-comp compare } = (<) \]

\[ \text{compare-order is compare and linorder, where comparator and orders define the same ordering.} \]

\textbf{subclass (in compare-order) linorder}
\textit{proof}

\textbf{context compare-order}
\textbf{begin}

\textbf{lemma compare-is-comparator-of:}
\[ \text{compare } = \text{comparator-of} \]
\textit{proof}

\textbf{lemmas two-comparisons-into-compare =}
\[ \text{comparator.two-comparisons-into-case-order[OF comparator-compare, unfolded ord-defs]} \]

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

\textit{ML}

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

\textbf{end}

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

\textbf{theory RBT-Compare-Order-Impl}
\textbf{imports}
\texttt{Compare}
\texttt{HOL-Library.RBT-Impl}
\textbf{begin}

\textbf{end}
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.

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

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

3.4 A Comparator-Interface to Red-Black-Trees

\texttt{theory RBT-Comparator-Impl}
\texttt{imports HOL-Library.RBT-Impl}
\texttt{Comparator}
\texttt{begin}

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

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

\texttt{context}
\texttt{fixes c :: 'a comparator}
\texttt{begin}

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

\texttt{fun rbt-comp-ins :: ('a ⇒ 'b ⇒ 'b ⇒ 'b ⇒ 'a ⇒ 'b ⇒ 'a,'b) rbt ⇒ ('a,'b) rbt}
\texttt{where}
\texttt{rbt-comp-ins f k v RBT-Impl.Empty = Branch RBT-Impl.R RBT-Impl.Empty k v RBT-Impl.Empty |}
primrec
y s

and

fun
definition

rbt-comp-insert-with-key f k v t
\hspace{1cm} = \hspace{1cm} rbt-comp-del-from-left x a y s b
\hspace{1cm} rbt-comp-map-entry k f RBT-Impl
\hspace{1cm} ::
\hspace{1cm} rbt-comp-del x
\hspace{1cm} \Rightarrow \hspace{1cm} Gt
\hspace{1cm} \Rightarrow \hspace{1cm} Lt
\hspace{1cm} \Rightarrow \hspace{1cm} Lt
\hspace{1cm} \Rightarrow \hspace{1cm} Gt
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\hspace{1cm} \Rightarrow \hspace{1cm} R
\ hs
\textbf{function} \textit{comp-union-with} :: \( (\lambda a \Rightarrow \lambda b \Rightarrow \lambda b' \Rightarrow b') \Rightarrow (\lambda a \times \lambda b) \Rightarrow (\lambda a \times \lambda b) \)  
\textit{list} \Rightarrow (\lambda a \times \lambda b) \textit{list}  
\textbf{where}  
\textit{comp-union-with} \ f \ ((k, v) \ # \ as) \ ((k', v') \ # \ bs) =  
\hspace{1em} (\text{case } c \ k \ k \ \text{of} \  
\hspace{2em} \text{Lt} \Rightarrow (k', v') \ # \ \text{comp-union-with} \ f \ ((k, v) \ # \ as) \ bs  
\hspace{2em} \text{Gt} \Rightarrow (k, v) \ # \ \text{comp-union-with} \ f \ as \ ((k', v') \ # \ bs)  
\hspace{2em} \text{Eq} \Rightarrow (k, f k v v') \ # \ \text{comp-union-with} \ f \ as \ bs)  
\textbf{termination} \ \langle \text{proof} \rangle  

\textbf{function} \textit{comp-inter-with} :: \( (\lambda a \Rightarrow \lambda b \Rightarrow \lambda b' \Rightarrow b') \Rightarrow (\lambda a \times \lambda b) \Rightarrow (\lambda a \times \lambda b) \)  
\textit{list} \Rightarrow (\lambda a \times \lambda b) \textit{list}  
\textbf{where}  
\textit{comp-inter-with} \ f \ ((k, v) \ # \ as) \ ((k', v') \ # \ bs) =  
\hspace{1em} (\text{case } c \ k \ k \ \text{of} \  
\hspace{2em} \text{Lt} \Rightarrow \text{comp-inter-with} \ f \ ((k, v) \ # \ as) \ bs  
\hspace{2em} \text{Gt} \Rightarrow \text{comp-inter-with} \ f \ as \ ((k', v') \ # \ bs)  
\hspace{2em} \text{Eq} \Rightarrow (k, f k v v') \ # \ \text{comp-inter-with} \ f \ as \ bs)  
\textbf{termination} \ \langle \text{proof} \rangle  

\textbf{definition} \textit{rbt-comp-union-with-key} :: \( (\lambda a \Rightarrow \lambda b \Rightarrow \lambda b' \Rightarrow b') \Rightarrow (\lambda a, \lambda b) \Rightarrow (\lambda a, \lambda b) \)  
\textit{rbt} \Rightarrow (\lambda a, \lambda b) \textit{rbt}  
\textbf{where}  
\textit{rbt-comp-union-with-key} \ f \ t1 \ t2 =  
\hspace{1em} (\text{case } \text{RBT-Impl.compare-height} \ t1 \ t1 \ t2 \ t2  
\hspace{2em} \text{of } \text{compare.EQ} \Rightarrow \text{rbtreeify} \ (\text{comp-union-with} \ f \ (\text{RBT-Impl.entries} \ t1) \ (\text{RBT-Impl.entries} \ t2))  
\hspace{2em} \text{| compare.LT} \Rightarrow \text{RBT-Impl.fold} \ (\lambda a \times \lambda b \Rightarrow \text{rbt-comp-insert-with-key} \ (\lambda k v w. f k v w)) \ t1 \ t2  
\hspace{2em} \text{| compare.GT} \Rightarrow \text{RBT-Impl.fold} \ (\lambda a \times \lambda b \Rightarrow \text{rbt-comp-insert-with-key} \ f) \ t2 \ t1)  

\textbf{definition} \textit{rbt-comp-inter-with-key} :: \( (\lambda a \Rightarrow \lambda b \Rightarrow \lambda b' \Rightarrow b') \Rightarrow (\lambda a, \lambda b) \Rightarrow (\lambda a, \lambda b) \)  
\textit{rbt} \Rightarrow (\lambda a, \lambda b) \textit{rbt}  
\textbf{where}  
\textit{rbt-comp-inter-with-key} \ f \ t1 \ t2 =  
\hspace{1em} (\text{case } \text{RBT-Impl.compare-height} \ t1 \ t1 \ t2 \ t2  
\hspace{2em} \text{of } \text{compare.EQ} \Rightarrow \text{rbtreeify} \ (\text{comp-inter-with} \ f \ (\text{RBT-Impl.entries} \ t1) \ (\text{RBT-Impl.entries} \ t2))  
\hspace{2em} \text{| compare.LT} \Rightarrow \text{rbtreeify} \ (\lambda a \times \lambda b \Rightarrow \text{rbtreeify} \ (\text{List.map-filter} \ (\lambda k v w. f k v w)) \ (\text{rbt-comp-lookup} \ t2 \ k)) \ (\text{RBT-Impl.entries} \ t1))  
\hspace{2em} \text{| compare.GT} \Rightarrow \text{rbtreeify} \ (\lambda a \times \lambda b \Rightarrow \text{rbtreeify} \ (\text{List.map-filter} \ (\lambda k v w. f k v w)) \ (\text{rbt-comp-lookup} \ t1 \ k)) \ (\text{RBT-Impl.entries} \ t2))  

10
context
  assumes $c$: comparator $c$
begin

lemma $\text{rbt-comp-lookup}$: $\text{rbt-comp-lookup} = \text{ord.rbttlookup} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{rbt-comp-ins}$: $\text{rbt-comp-ins} = \text{ord.rbttins} (\text{lt-of-comp } c)$
⟨proof⟩

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

lemma $\text{rbt-comp-insert}$: $\text{rbt-comp-insert} = \text{ord.rbttinsert} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{rbt-comp-del}$: $\text{rbt-comp-del} = \text{ord.rbttdel} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{rbt-comp-delete}$: $\text{rbt-comp-delete} = \text{ord.rbttdelete} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{rbt-comp-bulkload}$: $\text{rbt-comp-bulkload} = \text{ord.rbttbulkload} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{rbt-comp-map-entry}$: $\text{rbt-comp-map-entry} = \text{ord.rbttmap-entry} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{comp-sunion-with}$: $\text{comp-sunion-with} = \text{ord.sunion-with} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{comp-sinter-with}$: $\text{comp-sinter-with} = \text{ord.sinter-with} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{rbt-comp-union-with-key}$: $\text{rbt-comp-union-with-key} = \text{ord.rbttunion-with-key} (\text{lt-of-comp } c)$
⟨proof⟩

lemma $\text{rbt-comp-inter-with-key}$: $\text{rbt-comp-inter-with-key} = \text{ord.rbttinter-with-key} (\text{lt-of-comp } c)$
⟨proof⟩

lemmas $\text{rbt-comp-simps} =$
  $\text{rbt-comp-insert}$
  $\text{rbt-comp-lookup}$
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) | z ⇒ z)
(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 = Eq \iff (∀ ord ∈ set os. ord = Eq)

(Proof)

**Definition** trans-order :: order ⇒ order ⇒ order ⇒ bool where
trans-order x y z \iff x ≠ Gt \land y ≠ Gt \land (x = Lt \lor y = Lt) \implies z = Lt

**Lemma** trans-orderI:
(x ≠ Gt \implies y ≠ Gt \implies z ≠ Gt \land ((x = Lt \lor y = Lt) \implies z = Lt)) \implies
trans-order x y z
(Proof)

**Lemma** trans-orderD:
assumes trans-order x y z and x ≠ Gt and y ≠ Gt
shows z ≠ Gt and x = Lt \lor y = Lt \implies z = Lt
(Proof)

**Lemma** All-less-Suc:
(∀ i < Suc x. P i) \iff P 0 \land (∀ i < x. P (Suc i))
(Proof)

**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)
sshows trans-order (comp-lex xs) (comp-lex ys) (comp-lex zs)
(Proof)

**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
(Proof)

declare comp-lex.simps [simp del]

**Definition** peq-comp :: 'a comparator ⇒ 'a ⇒ bool
where
peq-comp acomp x \iff (∀ y. acomp x y = Eq \iff x = y)

**Lemma** peq-compD: peq-comp acomp x \implies acomp x y = Eq \iff x = y
(Proof)
lemma peq-compI: \( (\forall y. \text{acomp} \ x \ y = \text{Eq} \iff x = y) \implies \text{peq-comp} \ \text{acomp} \ x \)  
\( \langle \text{proof} \rangle \)

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

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

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

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

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

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

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

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

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

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

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

lemma trans-compI2: \( (\forall x. \text{ptrans-comp} \ \text{acomp} \ x) \implies \text{trans-comp} \ \text{acomp} \)  
\( \langle \text{proof} \rangle \)
definition sym-comp :: 'a comparator ⇒ bool where
    sym-comp acomp ←→ (∀ x. psym-comp acomp x)

lemma sym-compD\': sym-comp acomp ⇒ psym-comp acomp x
⟨proof⟩

lemma sym-compI\': (∀ 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
stance 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 \( prod :: (\text{compare}, \text{compare}) \text{ compare-order} \).

• **derive linorder prod** creates an instance \( prod :: (\text{linorder}, \text{linorder}) \text{ linorder} \).

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

**lemma linorder-axiomsD**: assumes class.linorder le lt
shows
\[
\begin{align*}
\text{lt } x y &= (\text{le } x y \land \neg \text{le } y x) \quad (\text{is } \texttt{?a}) \\
\text{le } x x &= (\text{is } \texttt{?b}) \\
\text{le } x y \implies \text{le } y z &\implies \text{le } x z \quad (\text{is } \texttt{?c1} \implies \texttt{?c2} \implies \texttt{?c3}) \\
\text{le } x y \implies \text{le } y x &\implies x = y \quad (\text{is } \texttt{?d1} \implies \texttt{?d2} \implies \texttt{?d3}) \\
\text{le } x y \lor \text{le } y x &= (\text{is } \texttt{?e})
\end{align*}
\]
(\text{proof})

**named-theorems** compare-simps simp theorems to derive compare = comparator-of

(\text{ML})

end

4.8 Defining Comparators and Compare-Instances for Common Types

**theory** Compare-Instances
**imports**
  Compare-Generator
  HOL-Library.Char-ord
**begin**

For all of the following types, we define comparators and register them in the class compare: \( \text{int}, \text{integer}, \text{nat}, \text{char}, \text{bool}, \text{unit}, \text{sum}, \text{option}, \text{list}, \) and \( \text{prod} \). 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 \( \text{int}, \text{nat}, \text{integer} \) and \( \text{char} \) we just use their linear orders as comparators.

**derive** (linorder) compare-order int integer nat char
For `sum`, `list`, `prod`, and `option` we generate comparators which are however are not used to instantiate `linorder`.

**derive** `compare sum list prod option`

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

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

**fun** `comparator-bool :: bool comparator` where
`comparator-bool False False = Eq`
`positive comparator-bool False True = Lt`
`positive comparator-bool True True = Eq`
`positive 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]:`
`positive comparator-bool = comparator-of` (proof)

**lemma** `comparator-unit-comparator-of [compare-simps]:`
`positive 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`
`HOL-Library.List-Lexorder`
`HOL-Library.Product-Lexorder`
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.

```plaintext
derive compare-order sum

instance list :: (compare-order)compare-order (proof)
instance prod :: (compare-order, compare-order)compare-order (proof)
instance option :: (compare-order)compare-order (proof)
end
```

### 4.10 Compare Instance for Rational Numbers

```plaintext
theory Compare-Rat
imports HOL.Rat
begin

derive (linorder) compare-order rat
end
```

### 4.11 Compare Instance for Real Numbers

```plaintext
theory Compare-Real
imports HOL.Real
begin

derive (linorder) compare-order real

lemma invert-order-comp-real[simp]: \( \forall x\ y :: real. \text{invert-order} (\text{compare} x\ y) = \text{compare} y\ x \) (proof)
end
```
5 Checking Equality Without ”=”

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

typedecl ('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))
⟨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 \implies aeq x y \iff x = y
(\langle \text{proof} \rangle)

lemma pequalityI: (\forall y. aeq x y \iff x = y) \implies pequality aeq x
(\langle \text{proof} \rangle)

5.3 Global equality property

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

lemma equalityD2: equality aeq \implies \text{pequality} aeq x
(\langle \text{proof} \rangle)

lemma equalityI2: (\forall x. \text{pequality} aeq x) \implies equality aeq
(\langle \text{proof} \rangle)

lemma equalityD: equality aeq \implies aeq x y \iff x = y
(\langle \text{proof} \rangle)

lemma equalityI: (\forall x y. aeq x y \iff x = y) \implies equality aeq
(\langle \text{proof} \rangle)

lemma equality-imp-eq:
equality aeq \implies aeq = (=)
(\langle \text{proof} \rangle)

lemma eq-equality: equality (=)
(\langle \text{proof} \rangle)

lemma equality-def': equality f = (f = (=))
(\langle \text{proof} \rangle)

5.4 The Generator
(\langle ML \rangle)

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. int, integer, nat, char, bool, unit, sum, option, list, and prod. For types without type
parameters, we use plain (\(=\)), and for the others we use generated ones. These functions will be essential, when the generator is later on invoked on types, which in their definition use one these types.

\[
\text{derive (eq) equality int integer nat char bool unit}
\]

\[
\text{derive equality sum list prod option}
\]

end

6 Generating Hash-Functions

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

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

\[
\begin{align*}
\text{fun hash-combine :: hashcode list } & \Rightarrow \text{ hashcode list } \Rightarrow \text{ hashcode where } \\
& \text{ hash-combine } [] [x] = x \\
& | \text{ hash-combine } (y # ys) (z # zs) = y \times z + \text{ hash-combine } ys zs \\
& | \text{ hash-combine } - - = 0
\end{align*}
\]

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 magic numbers which are generated within the generator.

6.1 Improved Code for Non-Lazy Languages

\[
\text{lemma hash-combine-unfold:}
\]

\[
\begin{align*}
\text{hash-combine } [] [x] &= x \\
\text{hash-combine } (y # ys) (z # zs) &= y \times z + \text{ hash-combine } ys zs
\end{align*}
\]

\[
\text{⟨proof⟩}
\]

6.2 The Generator

\[
\langle ML \rangle
\]

end

6.3 Defining Hash-Functions for Common Types

theory Has-Instances
imports
Hash-Generator
For all of the following types, we register hashcode-functions. \textit{int, integer, nat, char, bool, unit, sum, option, list, and prod}. For types without type parameters, we use plain \textit{hashcode}, and for the others we use generated ones.

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

derive hash-code prod sum option list
\end{verbatim}

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

\begin{verbatim}
7 Countable Datatypes
theory Countable-Generator
imports
  HOL-Library.Countable
  ../Derive-Manager
begin

Brian Huffman and Alexander Krauss (old datatype), and Jasmin Blanchette (BNF datatype) have developed tactics which automatically can prove that a datatype is countable. We just make this tactic available in the derive-manager so that one can conveniently write \texttt{derive countable some-datatype}.

7.1 Installing the tactic
There is nothing more to do, then to write some boiler-plate ML-code for class-instantiation.
\begin{verbatim}
⟨ML⟩
end
\end{verbatim}

8 Loading Existing Derive-Commands
theory Derive
imports
  Comparator-Generator/Compare-Instances
  Equality-Generator/Equality-Instances
  Hash-Generator/Hash-Instances
  Countable-Generator/Countable-Generator
begin
We just load the commands to derive comparators, equality-functions, hash-functions, and the command to show that a datatype is countable, so that now all of them are available. There are further generators available in the AFP entries Containers and Show.

print-derives

end

9 Examples

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

9.1 Rational Numbers

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

Use the linear order on rationals to define the \texttt{compare-order}-instance.

\begin{verbatim}
derive (linorder) compare-order rat
\end{verbatim}

Use \texttt{(=)} as equality function.

\begin{verbatim}
derive (eq) equality rat
\end{verbatim}

First manually define a hashcode function.

\begin{verbatim}
instantiation rat :: hashable
begin
  definition def-hashmap-size = (\lambda - :: rat itself. 10)
  definition hashcode (r :: rat) = hashcode (quotient-of r)
instance
  ⟨proof⟩
end
\end{verbatim}

And then register it at the generator.

\begin{verbatim}
derive (hashcode) hash-code rat
\end{verbatim}

9.2 A Datatype Without Nested Recursion

datatype 'a bintree = BEmpty | BNode 'a bintree 'a 'a bintree
9.3 Using Other datatypes

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

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

9.4 Mutual Recursion

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

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

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

9.5 Nested recursion

datatype 'a tree = Empty | Node 'a 'a tree list
data type '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

data type ('f, 'v) term = Var 'v | Fun 'f ('f,'v) term list

data type ('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
9.7 A Complex Datatype

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

\begin{verbatim}
data datatype ('a, 'b) complex =  
C1 nat 'a ttree × rat + ('a,'b) complex list |  
C2 ('a, 'b) complex list tree tree 'b ('a, 'b) complex ('a, 'b) complex2 ttree list  
and ('a, 'b) complex2 = D1 ('a, 'b) complex ttree  
end
\end{verbatim}

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.

\begin{verbatim}
derive equality complex  
derive hashable complex complex2  
derive countable complex complex2  
derive equality complex  
derive hashable complex complex2  
end
\end{verbatim}

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

