Abstract

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

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

Our formalization was performed as part of the IsaFoR/CeTA project\(^1\) [3]. With our new tactics we could remove several tedious proofs for (conditional) linear orders, and conditional equality operators within IsaFoR and the Container Framework.

Contents

1 Derive Manager

2 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.
\(^1\)http://cl-informatik.uibk.ac.at/software/ceta
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.

\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 datatype \textit{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.

\langle ML \rangle

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

\langle ML \rangle

lemma in-set-simps:
  \( x \in \text{set} \ (y \# z \# y) = (x = y \lor x \in \text{set} \ (z \# ys)) \)
  \( x \in \text{set} \ ([y]) = (x = y) \)
  \( x \in \text{set} \ [\ ] = \text{False} \)
  \( \text{Ball} \ (\text{set} \ [\ ]) \ P = \text{True} \)
  \( \text{Ball} \ (\text{set} \ [x]) \ P = P x \)

\langle ML \rangle
Ball \((set (x \neq y \neq zs)) P = (P x \land Ball (set (y \neq zs)) P)\)

\(\langle proof \rangle\)

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

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, \(((<), (\leq))\), one can alternative use a comparator to define the linear order, which may deliver three possible outcomes when comparing two values.

datatype order = Eq |Lt |Gt

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 
\(\langle proof \rangle\)

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: \(\bigwedge x y. lt x y \implies le x y \implies le-of-comp (comp-of-ords le lt) = le\) 
\(\langle proof \rangle\)

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))
(if le x y then (if lt x y then Q else P) else R) = (case-order P Q R (comp x y))
(if lt x y then Q else (if le x y then P else R)) = (case-order P Q R (comp x y))
(if lt x y then Q else (if lt y x then R else P)) = (case-order P Q R (comp x y))
(if lt x y then Q else (if x = y then P else R)) = (case-order P Q R (comp x y))
(if 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 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))

(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
begin

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

lemma compare-refl [simp]:
\begin{verbatim}
compare x x = Eq
⟨proof⟩
end

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

class compare-order = ord + compare +
assumes ord-defs: le-of-comp compare = (≤) lt-of-comp compare = (<)

  compare-order is compare and linorder, where comparator and orders
define the same ordering.

subclass (in compare-order) linorder
⟨proof⟩

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 equa-
tions of some constant such that two consecutive comparisons via (≤), (<)”,
or (=) are turned into one invocation of compare. The difference to a stan-
dard code-unfold is that here we change the code-equations where an ad-
ditional 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
  HOL-Library.RBT-Impl
begin

end
\end{verbatim}
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`.

```plaintext
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) sint-with

export-code rbt-ins rbt-lookup rbt-del rbt-map-entry rbt-union-with-key rbt-inter-with-key
in Haskell
```

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

**theory** RBT-Comparator-Impl

**imports**

`HOL-Library.RBT-Impl Comparator`

**begin**

For all of the main algorithms of red-black trees, we provide alternatives which are completely based on comparators, and which are 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) rbt ⇒ ('a,'b) rbt

**where**

```
rbt-comp-ins f k v RBT-Impl.Empty = Branch RBT-Impl.R RBT-Impl.Empty k
v RBT-Impl.Empty |
```

8
function \texttt{comp-sumion-with} :: \texttt{('a ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a × 'b) list}

\textbf{where}

\texttt{comp-sumion-with f ((k, v) \# as) ((k', v') \# bs) =}
(case c k' k of
  Lt ⇒ \texttt{comp-sumion-with f ((k', v') \# as) bs}
  | Gt ⇒ \texttt{comp-sumion-with f as ((k', v') \# bs)}
  | Eq ⇒ \texttt{comp-sumion-with f as bs})

\texttt{proof}

termination \texttt{(proof)}

function \texttt{comp-sinter-with} :: \texttt{('a ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a × 'b) list}

\textbf{where}

\texttt{comp-sinter-with f ((k, v) \# as) ((k', v') \# bs) =}
(case c k' k of
  Lt ⇒ \texttt{comp-sinter-with f ((k', v') \# as) bs}
  | Gt ⇒ \texttt{comp-sinter-with f as ((k', v') \# bs)}
  | Eq ⇒ \texttt{comp-sinter-with f as bs})

\texttt{proof}

termination \texttt{(proof)}

definition \texttt{rbd-comp-union-with-key} :: \texttt{('a ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbd ⇒ ('a, 'b) rbd}

\textbf{where}

\texttt{rbd-comp-union-with-key f t1 t2 =}
(case RBT-Impl.compare-height t1 t1 t2 t2
  of compare.EQ ⇒ \texttt{rbtreeify (comp-sumion-with f (RBT-Impl.entries t1) (RBT-Impl.entries t2))}
  | compare.LT ⇒ RBT-Impl.fold (rbd-comp-insert-with-key (λk v w. f k v w)) t1 t2
  | compare.GT ⇒ RBT-Impl.fold (rbd-comp-insert-with-key f) t2 t1)

definition \texttt{rbd-comp-inter-with-key} :: \texttt{('a ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbd ⇒ ('a, 'b) rbd}

\textbf{where}

\texttt{rbd-comp-inter-with-key f t1 t2 =}
(case RBT-Impl.compare-height t1 t1 t2 t2
  of compare.EQ ⇒ \texttt{rbtreeify (comp-sinter-with f (RBT-Impl.entries t1) (RBT-Impl.entries t2))}
  | compare.LT ⇒ \texttt{rbtreeify (List.map-filter (λ(k, v). map-option (λw. (k, f k v w)) (rbd-comp-lookup t2 k)) (RBT-Impl.entries t1))}
  | compare.GT ⇒ \texttt{rbtreeify (List.map-filter (λ(k, v). map-option (λw. (k, f k v w)) (rbd-comp-lookup t1 k)) (RBT-Impl.entries t2))})
context
  assumes c: comparator c
begin

lemma rbt-comp-lookup: rbt-comp-lookup = ord.rbt-lookup (lt-of-comp c)
⟨proof⟩

lemma rbt-comp-ins: rbt-comp-ins = ord.rbt-ins (lt-of-comp c)
⟨proof⟩

(lt-of-comp c)
⟨proof⟩

lemma rbt-comp-insert: rbt-comp-insert = ord.rbt-insert (lt-of-comp c)
⟨proof⟩

lemma rbt-comp-del: rbt-comp-del = ord.rbt-del (lt-of-comp c)
⟨proof⟩

lemma rbt-comp-delete: rbt-comp-delete = ord.rbt-delete (lt-of-comp c)
⟨proof⟩

lemma rbt-comp-bulkload: rbt-comp-bulkload = ord.rbt-bulkload (lt-of-comp c)
⟨proof⟩

lemma rbt-comp-map-entry: rbt-comp-map-entry = ord.rbt-map-entry (lt-of-comp c)
⟨proof⟩

lemma comp-sunion-with: comp-sunion-with = ord.sunion-with (lt-of-comp c)
⟨proof⟩

lemma comp-sinter-with: comp-sinter-with = ord.sinter-with (lt-of-comp c)
⟨proof⟩

lemma rbt-comp-union-with-key: rbt-comp-union-with-key = ord.rbt-union-with-key
(lt-of-comp c)
⟨proof⟩

lemma rbt-comp-inter-with-key: rbt-comp-inter-with-key = ord.rbt-inter-with-key
(lt-of-comp c)
⟨proof⟩

lemmas rbt-comp-simps =
  rbt-comp-insert
  rbt-comp-lookup
4 Generating Comparators

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

typedecl (′a,′b,′c,′z)type

  In the following, we define a generator which for a given datatype (′a, 
  ′b, ′c, ′z) Comparator-Generator.type constructs a comparator of type ′a 
  comparator ⇒ ′b comparator ⇒ ′c comparator ⇒ ′z comparator ⇒ (′a, ′b, 
  ′c, ′z) Comparator-Generator.type. To this end, we first compare the index 
  of the constructors, then for equal constructors, we compare the arguments 
  recursively and combine the results lexicographically.

hide-type type

4.1 Lexicographic combination of order

fun comp-lex :: order list ⇒ order
where
  comp-lex (c ≠ cs) = (case c of Eq ⇒ comp-lex cs | - ⇒ c) |
  comp-lex [] = Eq

4.2 Improved code for non-lazy languages

The following equations will eliminate all occurrences of comp-lex in the 
generated code of the comparators.

lemma comp-lex-unfolds:
  comp-lex [] = Eq
  comp-lex [c] = c
  comp-lex (c ≠ d ≠ cs) = (case c of Eq ⇒ comp-lex (d ≠ cs) | z ⇒ z)
(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 ←→ (∀ ord ∈ set os. ord = Eq)

**proof**

**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

**proof**

**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

**proof**

**lemma** All-less-Suc:

(∀ i < Suc x. P i) =⇒ P 0 ∧ (∀ 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)

shows 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 ←→ (∀ y. acomp x y = Eq ←→ x = y)

**lemma** peq-compD: peq-comp acomp x =⇒ acomp x y = Eq =⇒ x = y

**proof**
lemma peq-compI: \((\forall y. acomp x y = Eq \iff x = y) \implies peq-comp acomp x\)
(proof)

definition psym-comp :: 'a comparator \Rightarrow 'a \Rightarrow bool where
psym-comp acomp x \iff (\forall y. 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)
(proof)

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

definition ptrans-comp :: 'a comparator \Rightarrow 'a \Rightarrow bool where
ptrans-comp acomp x \iff (\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)
(proof)

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

4.4 Separate properties of comparators

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

lemma eq-compD2: eq-comp acomp \implies peq-comp acomp x
(proof)

lemma eq-compI2: (\forall x. peq-comp acomp x) \implies eq-comp acomp
(proof)

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

lemma trans-compD2: trans-comp acomp \implies ptrans-comp acomp x
(proof)

lemma trans-compI2: (\forall x. ptrans-comp acomp x) \implies trans-comp acomp

14
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: \textit{psym-comp} acomp \ x
\langle proof \rangle

lemma comp-to-peq-comp: \textit{peq-comp} acomp \ x
\langle proof \rangle

lemma comp-to-ptrans-comp: \textit{ptrans-comp} acomp \ x
\langle proof \rangle

\end

4.5 Auxiliary Lemmas for Comparator Generator

\begin{itemize}
\item lemma \textit{forall-finite}: \((\forall \ i < (0 :: \textit{nat}. \ P \ i) = \text{True}
\langle proof \rangle

4.6 The Comparator Generator

\begin{itemize}
\item \texttt{ML}
\end{itemize}

4.7 Compare Generator

theory \textit{Compare-Generator}
imports
\textit{Comparator-Generator}
\textit{Compare}
begin

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

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

\begin{itemize}
\item \texttt{derive compare prod} creates an instance \texttt{prod :: (compare, compare)} \texttt{compare}.
\end{itemize}
• **derive compare-order prod** creates an instance \( \text{prod} :: (\text{compare}, \text{compare}) \text{compare-order} \).

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

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
\[
\begin{align*}
\text{lt} \; x \; y & = (\text{le} \; x \; y \land \neg \text{le} \; y \; x) \; (\text{is } ?a) \\
\text{le} \; x \; x & (\text{is } ?b) \\
\text{le} \; x \; y \implies \text{le} \; y \; z & \implies \text{le} \; x \; z \; (\text{is } ?c1 \implies ?c2 \implies ?c3) \\
\text{le} \; x \; y \implies \text{le} \; y \; x & \implies x = y \; (\text{is } ?d1 \implies ?d2 \implies ?d3) \\
\text{le} \; x \; y \lor \text{le} \; y \; x & (\text{is } ?e)
\end{align*}
\]

\langle \text{proof} \rangle
\langle \text{ML} \rangle

**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 \( \text{compare} \): \( \text{int} \), \( \text{integer} \), \( \text{nat} \), \( \text{char} \), \( \text{bool} \), \( \text{unit} \), \( \text{sum} \), \( \text{option} \), \( \text{list} \), and \( \text{prod} \). We do not register those classes in \( \text{compare-order} \) 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** \((\text{linorder}) \text{compare-order int integer nat char}\)

For \( \text{sum} \), \( \text{list} \), \( \text{prod} \), and \( \text{option} \) we generate comparators which are however are not used to instantiate \( \text{linorder} \).
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
HOL-Library.List-Lexorder
HOL-Library.Product-Lexorder
HOL-Library.Option-ord
begin
We now also instantiate class \texttt{compare-order} and not only \texttt{compare}. Here, we also prove that our definitions do not clash with existing orders on \texttt{list}, \texttt{option}, and \texttt{prod}.

For \texttt{sum} we just define the linear orders via their comparator.

\begin{verbatim}
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
\end{verbatim}

4.10 Compare Instance for Rational Numbers

\begin{verbatim}
theory Compare-Rat
imports
  Compare-Generator
  HOL.Rat
begin

derive (linorder) compare-order rat

end
\end{verbatim}

4.11 Compare Instance for Real Numbers

\begin{verbatim}
theory Compare-Real
imports
  Compare-Generator
  HOL.Real
begin

derive (linorder) compare-order real

lemma invert-order-compare-real[simp]: \( \forall \ x \ y :: \text{real}. \ \text{invert-order} \ (\text{compare} \ x \ y) = \text{compare} \ y \ x \) ⟨proof⟩

end
\end{verbatim}
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 \rightarrow aeq x y \iff x = y
⟨proof\rangle

lemma pequalityI: (\forall y. aeq x y \iff x = y) \Rightarrow pequality aeq x
⟨proof\rangle

5.3 Global equality property

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

lemma equalityD2: equality aeq \Rightarrow pequality aeq x
⟨proof\rangle

lemma equalityI2: (\forall x. pequality aeq x) \Rightarrow equality aeq
⟨proof\rangle

lemma equalityD: equality aeq \Rightarrow aeq x y \iff x = y
⟨proof\rangle

lemma equalityI: (\forall x y. aeq x y \iff x = y) \Rightarrow equality aeq
⟨proof\rangle

lemma equality-imp-eq:
equality aeq \Rightarrow aeq = (=)
⟨proof\rangle

lemma eq-equality: equality (=)
⟨proof\rangle

lemma equality-def': equality f = (f = (=))
⟨proof\rangle

5.4 The Generator
⟨ML⟩

hide-fact (open) equalityI

end

5.5 Defining Equality-Functions for Common Types

theory Equality-Instances
imports
  Equality-Generator
begin

For all of the following types, we register equality-functions. \textit{int}, \textit{integer},
\textit{nat}, \textit{char}, \textit{bool}, \textit{unit}, \textit{sum}, \textit{option}, \textit{list}, and \textit{prod}. For types without type
parameters, we use plain (\(=\)), 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 Hash-Generator imports
..Generator-Aux
..Derive-Manager
Collections.HashCode

begin

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

fun hash-combine :: hashcode list => hashcode list => hashcode where
  hash-combine [] [x] = x
  | hash-combine (y # ys) (z # zs) = y * z + hash-combine ys zs
  | hash-combine - - = 0

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

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

\[\text{proof}\]

6.2 The Generator

\[\text{ML}\]

end

6.3 Defining Hash-Functions for Common Types

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

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

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

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

\section{Countable Datatypes}

\texttt{theory Countable-Generator}
\texttt{imports \HOL\Library.Countable 
../Derive-Manager}
\texttt{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}.

\section{Installing the tactic}

There is nothing more to do, then to write some boiler-plate ML-code for 
class-instantiation.

\begin{verbatim}
end
\end{verbatim}

8 Loading Existing Derive-Commands

\texttt{theory Derive}
\texttt{imports 
 Comparator-Generator/Compare-Instances 
 Equality-Generator/Equality-Instances 
 Hash-Generator/Hash-Instances 
 Countable-Generator/Countable-Generator}
\texttt{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 compare-order-instance.

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

Use (=) as equality function.

```
derive (eq) equality rat
```

First manually define a hashcode function.

```
instantiation rat :: hashable
begin
definition def-hashmap-size = (λ- :: rat itself. 10)
definition hashcode (r :: rat) = hashcode (quotient-of r)
instance
  ⟨proof⟩
end

And then register it at the generator.

```
derive (hashcode) hash-code rat
```

9.2 A Datatype Without Nested Recursion

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

24
derive compare-order bintree
derive countable bintree
derive equality bintree
derive hashable bintree

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

9.4 Mutual Recursion
datatype 'a mtree = MEmpty | MNode 'a 'a mtree-list and
'a mtree-list = MNil | MCons 'a mtree 'a mtree-list
derive compare-order mtree mtree-list
derive countable mtree mtree-list
derive hashable mtree mtree-list

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

9.5 Nested recursion
datatype 'a tree = Empty | Node 'a 'a tree list
datatype 'a ttree = TEmpty | TNode 'a 'a ttree list tree
derive compare-order tree ttree
derive countable tree ttree
derive equality tree ttree
derive hashable tree ttree

9.6 Examples from IsaFoR
datatype ('f,'v) term = Var 'v | Fun 'f ('f,'v) term list
datatype ('f,'l) lab =
  Lab ('f,'l) lab 'l
| FunLab ('f,'l) lab ('f,'l) lab list
| UnLab 'f
| 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 × rat + ('a,'b) complex list |
  C2 ('a, 'b) complex list tree tree 'b ('a, 'b) complex ('a, 'b) complex2 ttree list

and ('a, 'b) complex2 = D1 ('a, 'b) complex ttree

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

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

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

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

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

