Deriving class instances for datatypes.*

Christian Sternagel and René Thiemann

September 19, 2015

Abstract

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

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

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

Contents

1 Derive Manager

2 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
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 12
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

5 Checking Equality Without ”=”
5.1 Improved Code for Non-Lazy Languages .......................... 19
5.2 Partial Equality Property ....................................... 20
5.3 Global equality property ....................................... 20
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 ................................. 23

9 Examples
9.1 Rational Numbers ............................................... 23
9.2 A Datatype Without Nested Recursion .......................... 24
9.3 Using Other datatypes .......................................... 24
9.4 Mutual Recursion ............................................... 24
9.5 Nested recursion ............................................... 24
9.6 Examples from IsaFoR ......................................... 25
9.7 A Complex Datatype ........................................... 25

10 Acknowledgements ................................................ 26

1 Derive Manager

theory Derive-Manager
imports Main
keywords print-derives :: diag and derive :: thy-decl
The derive manager allows the user to register various derive-hooks, e.g., for orders, pretty-printers, hash-functions, etc. All registered hooks are accessible via the derive command.

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

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

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

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

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

lemma conj-weak-cong: \( a = b \Rightarrow c = d \Rightarrow (a \land c) = (b \land d) \) \{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 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
  ⟨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 |

lemma refl-True: \((x = x) = True\) ⟨proof⟩

end
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 $\Rightarrow$ x = y
      and trans: comp x y = Lt $\Rightarrow$ comp y z = Lt $\Rightarrow$ 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 $\leftrightarrow$ lt y x
  ⟨proof⟩
lemma Lt-lt-conv: comp x y = Lt $\leftrightarrow$ lt x y
  ⟨proof⟩
lemma eq-Eq-conv: comp x y = Eq $\leftrightarrow$ x = y
  ⟨proof⟩
lemma nGt-le-conv: comp x y ¯= Gt $\leftrightarrow$ le x y
  ⟨proof⟩
lemma nLt-le-conv: comp x y ¯= Lt $\leftrightarrow$ le y x
  ⟨proof⟩
lemma nEq-neq-conv: comp x y ¯= Eq $\leftrightarrow$ 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 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))

5
\begin{align*}
\text{(if } \mathit{lt} \ x \ y \ \text{then } Q \ \text{else } (\text{if } y = x \ \text{then } P \ \text{else } R)) &= (\text{case-order } P \ Q \ R \ (\text{comp } x \ y)) \\
(\text{if } x = y \ \text{then } P \ \text{else } (\text{if } \mathit{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 } \mathit{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 } \mathit{le} \ 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 } \mathit{le} \ x \ y \ \text{then } Q \ \text{else } R)) &= (\text{case-order } P \ Q \ R \ (\text{comp } x \ y))
\end{align*}

\begin{proof}
\end{proof}

\textbf{lemma} \textit{comp-of-ords: assumes class.linorder le lt}
\begin{itemize}
\item \textbf{shows} comparator \textit{(comp-of-ords le lt)}
\end{itemize}
\begin{proof}
\end{proof}

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

\textbf{lemma} \textit{comparator-of: comparator comparator-of}
\begin{proof}
\end{proof}

\end{proof}

\section*{3.2 Compare}

\textbf{theory} \textit{Compare}
\begin{itemize}
\item \textbf{imports} Comparator
\item \textbf{keywords} compare-code :: thy-decl
\item \textbf{begin}
\end{itemize}

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 induced orders coincide.

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

\textbf{lemma} \textit{compare-Eq-is-eq [simp]}:
\begin{itemize}
\item \textit{compare} \textit{x y} = \textit{Eq} \leftrightarrow \textit{x} = \textit{y}
\end{itemize}
\begin{proof}
\end{proof}

\textbf{lemma} \textit{compare-refl [simp]}:
\begin{itemize}
\item \textit{compare} \textit{x x} = \textit{Eq}
\end{itemize}
\begin{proof}
\end{proof}

\end{proof}
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⟩

custom 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 \texttt{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) rbt-union-with-key
compare-code ('a) rbt-inter-with-key

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

\texttt{theory RBT-Comparator-Impl}

\texttt{imports ~~/src/HOL/Library/RBT-Impl 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 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.

\texttt{context}

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

\texttt{begin}

\texttt{primrec rbt-comp-lookup :: ('a, 'b) rbt \Rightarrow 'a \Rightarrow '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 \Rightarrow rbt-comp-lookup l k}
\texttt{| Gt \Rightarrow rbt-comp-lookup r k}
\texttt{| Eq \Rightarrow Some y)}

\texttt{fun rbt-comp-ins :: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('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}
\texttt{| rbt-comp-ins f k v (Branch RBT-Impl.B l x y r) = (case c k x of}
\texttt{Lt \Rightarrow balance (rbt-comp-ins f k v l) x y r}
\texttt{| Gt \Rightarrow balance l x y (rbt-comp-ins f k v r)}
| Eq ⇒ Branch RBT-Impl.B l x (f k y v) r | |
| rbt-comp-ins f k v (Branch RBT-Impl.R l x y r) = (case c k x of |
| Lt ⇒ Branch RBT-Impl.R (rbt-comp-ins f k v l) x y r |
| | Gt ⇒ Branch RBT-Impl.R l x y (rbt-comp-ins f k v r) |
| | Eq ⇒ Branch RBT-Impl.R l x (f k y v) r) |

**definition** rbt-comp-insert-with-key :: ('a ⇒ 'b ⇒ 'b ⇒ 'a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a ⇒ 'b) |

**where** rbt-comp-insert-with-key f k v t = paint RBT-Impl.B (rbt-comp-ins f k v t)

**definition** rbt-comp-insert :: 'a ⇒ 'b ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b) |

**where** rbt-comp-insert = rbt-comp-insert-with-key (λ- - νv. νv)

**fun** |
| rbt-comp-del-from-left :: ('a ⇒ ('a, 'b)) ⇒ ('a ⇒ 'b) ⇒ ('a, 'b) |

**and** |
| rbt-comp-del-from-right :: ('a ⇒ ('a, 'b)) ⇒ ('a ⇒ 'b) ⇒ ('a, 'b) |

**and** |
| rbt-comp-del :: ('a ⇒ ('a, 'b)) ⇒ ('a, 'b) |

**where** |
| rbt-comp-del x RBT-Impl.Empty = RBT-Impl.Empty |
| rbt-comp-del x (Branch - a y s b) = |
| (case c x y of |
| Lt ⇒ rbt-comp-del-from-left x a y s b |
| | Gt ⇒ rbt-comp-del-from-right x a y s b |
| | Eq ⇒ combine a b) |
| rbt-comp-del-from-left x (Branch RBT-Impl.B lt z v rt) y s b = balance-left |
| (rbt-comp-del x (Branch RBT-Impl.B lt z v rt)) y s b |
| rbt-comp-del-from-left x a y s b = Branch RBT-Impl.R (rbt-comp-del x a) y s b |
| rbt-comp-del-from-right x a y s (Branch RBT-Impl.B lt z v rt) = balance-right a |
| y s (rbt-comp-del x (Branch RBT-Impl.B lt z v rt)) |
| rbt-comp-del-from-right x a y s b = Branch RBT-Impl.R a y s (rbt-comp-del x b) |

**definition** rbt-comp-delete k t = paint RBT-Impl.B (rbt-comp-del k t)

**definition** rbt-comp-bulkload xs = foldr (λ(k, v). rbt-comp-insert k v) xs RBT-Impl.Empty

**primrec** |
| rbt-comp-map-entry :: ('a ⇒ ('b ⇒ 'b) ⇒ ('a ⇒ 'b) |

**where** |
| rbt-comp-map-entry k f RBT-Impl.Empty = RBT-Impl.Empty |
| (case c k x of |
| Lt ⇒ Branch cc (rbt-comp-map-entry k f lt) x v rt |
| | Gt ⇒ Branch cc lt x v (rbt-comp-map-entry k f rt) |
| | Eq ⇒ Branch cc lt x (f v) rt) |

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

list ⇒ ('a × 'b) list
where

\[ \text{comp-union-with } f \ ((k, v) \# as) ((k', v') \# bs) = \]
\[ (\text{case } c k' k \text{ of} \]
\[ \quad \text{Lt} \Rightarrow (k', v') \# \text{comp-union-with } f \ ((k, v) \# as) bs \]
\[ \quad \text{Gt} \Rightarrow (k, v) \# \text{comp-union-with } f \ as ((k', v') \# bs) \]
\[ \quad \text{Eq} \Rightarrow (k, f k v v') \# \text{comp-union-with } f \ as bs) \]
\[ \) comp-union-with \ f \ as \ [] = \ as \]
\) (proof) \]

termination (proof)

function \text{comp-sinter-with} :: (\text{'a} \Rightarrow \text{'b}) \Rightarrow (\text{'a} \times \text{'b}) \Rightarrow (\text{'a} \times \text{'b}) \Rightarrow (\text{'a} \times \text{'b}) \Rightarrow (\text{'a} \times \text{'b}) \Rightarrow (\text{'a} \times \text{'b}) \Rightarrow (\text{'a} \times \text{'b}) \Rightarrow (\text{'a} \times \text{'b}) \Rightarrow (\text{'a} \times \text{'b}) list \Rightarrow (\text{'a} \times \text{'b}) list

where

\[ \text{comp-sinter-with } f \ ((k, v) \# as) ((k', v') \# bs) = \]
\[ (\text{case } c k' k \text{ of} \]
\[ \quad \text{Lt} \Rightarrow \text{comp-sinter-with } f \ ((k, v) \# as) bs \]
\[ \quad \text{Gt} \Rightarrow \text{comp-sinter-with } f \ as ((k', v') \# bs) \]
\[ \quad \text{Eq} \Rightarrow (k, f k v v') \# \text{comp-sinter-with } f \ as bs) \]
\[ \) comp-sinter-with \ f \ as \ [] = \ [] \]
\) (proof) \]

termination (proof)

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

where

\[ \text{rbt-comp-union-with-key } f t1 t2 = \]
\[ (\text{case } RBT-Impl\text{-compare-height } t1 t1 t2 t2 \]
\[ \text{of } \text{compare.EQ} \Rightarrow \text{rbtreeify} \ (\text{comp-union-with } f \ (RBT-Impl\text{-entries } t1)) (RBT-Impl\text{-entries } t2)) \]
\[ \text{| compare.LT} \Rightarrow RBT-Impl\text{-fold} \ (\text{rbt-comp-insert-with-key} \ (\lambda k v w. f k w v)) \ t1 \]
\[ \text{| compare.GT} \Rightarrow RBT-Impl\text{-fold} \ (\text{rbt-comp-insert-with-key} \ f) \ t2 \]
\)

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

where

\[ \text{rbt-comp-inter-with-key } f t1 t2 = \]
\[ (\text{case } RBT-Impl\text{-compare-height } t1 t1 t2 t2 \]
\[ \text{of } \text{compare.EQ} \Rightarrow \text{rbtreeify} \ (\text{comp-sinter-with } f \ (RBT-Impl\text{-entries } t1)) (RBT-Impl\text{-entries } t2)) \]
\[ \text{| compare.LT} \Rightarrow \text{rbtreeify} \ (\text{List.map-filter} \ (\lambda k v w. f k w v) \ \text{map-option} \ (\lambda w. (k, f k v w)) \ (\text{rbt-comp-lookup } \ t2 \ k)) (RBT-Impl\text{-entries } t1)) \]
\[ \text{| compare.GT} \Rightarrow \text{rbtreeify} \ (\text{List.map-filter} \ (\lambda k v w. f k w v) \ \text{map-option} \ (\lambda w. (k, f k w v)) \ (\text{rbt-comp-lookup } \ t1 \ k)) (RBT-Impl\text{-entries } t2)) \)

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

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 =

\[
\begin{align*}
& \text{rbt-comp-insert} \\
& \text{rbt-comp-lookup} \\
& \text{rbt-comp-delete} \\
& \text{rbt-comp-bulkload} \\
& \text{rbt-comp-map-entry}
\end{align*}
\]
4 Generating Comparators

theory Comparator-Generator
imports../Generator-Aux../Derive-ManagerComparatorbegin

typedef ('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
lemma \textit{comp-lex-eq}: \textit{comp-lex}\ os = Eq \leftrightarrow (\forall \ ord \in \text{set os}.\ \text{ord} = \text{Eq})
\langle \text{proof} \rangle

definition \textit{trans-order} :: \text{order} \Rightarrow \text{order} \Rightarrow \text{order} \Rightarrow \text{bool} \where
\text{trans-order}\ x\ y\ z \leftrightarrow x \neq \text{Gt} \rightarrow y \neq \text{Gt} \rightarrow z \neq \text{Gt} \land ((x = \text{Lt} \lor y = \text{Lt}) \rightarrow z = \text{Lt})
\langle \text{proof} \rangle

lemma \textit{trans-orderI}:
(x \neq \text{Gt} \Rightarrow y \neq \text{Gt} \Rightarrow z \neq \text{Gt} \land ((x = \text{Lt} \lor y = \text{Lt}) \rightarrow z = \text{Lt})) \Rightarrow \text{trans-order}\ x\ y\ z
\langle \text{proof} \rangle

lemma \textit{trans-orderD}:
\text{assumes} \text{trans-order}\ x\ y\ z \and x \neq \text{Gt} \and y \neq \text{Gt}
\text{shows} \ z \neq \text{Gt} \and x = \text{Lt} \lor y = \text{Lt} \Rightarrow z = \text{Lt}
\langle \text{proof} \rangle

lemma \textit{All-less-Suc}:
(\forall i < \text{Suc}\ x.\ P\ i) \leftrightarrow P\ 0 \land (\forall i < x.\ P\ (\text{Suc}\ i))
\langle \text{proof} \rangle

lemma \textit{comp-lex-trans}:
\text{assumes} \text{length}\ xs = \text{length}\ ys \and \text{length}\ ys = \text{length}\ zs \and \forall i < \text{length}\ zs.\ \text{trans-order}\ (xs!i)\ (ys!i)\ (zs!i)
\text{shows} \text{trans-order}\ (\text{comp-lex}\ xs)\ (\text{comp-lex}\ ys)\ (\text{comp-lex}\ zs)
\langle \text{proof} \rangle

lemma \textit{comp-lex-sym}:
\text{assumes} \text{length}\ xs = \text{length}\ ys \and \forall i < \text{length}\ ys.\ \text{invert-order}\ (xs!i) = ys!i
\text{shows} \text{invert-order}\ (\text{comp-lex}\ xs) = \text{comp-lex}\ ys
\langle \text{proof} \rangle

declare \textit{comp-lex.simps} [\text{simp del}]

definition \textit{peq-comp} :: \'a\ \text{comparator} \Rightarrow \'a \Rightarrow \text{bool} \where
\text{peq-comp}\ \text{acomp}\ x \leftrightarrow (\forall y.\ \text{acomp}\ x\ y = \text{Eq} \leftrightarrow x = y)

lemma \textit{peq-compD}: \text{peq-comp}\ \text{acomp}\ x \Rightarrow \text{acomp}\ x\ y = \text{Eq} \leftrightarrow x = y
\langle \text{proof} \rangle

lemma \textit{peq-compI}:
(\forall y.\ \text{acomp}\ x\ y = \text{Eq} \leftrightarrow x = y) \Rightarrow \text{peq-comp}\ \text{acomp}\ x
\langle \text{proof} \rangle

definition \textit{psym-comp} :: \'a\ \text{comparator} \Rightarrow \'a \Rightarrow \text{bool} \where
\text{psym-comp}\ \text{acomp}\ x \leftrightarrow (\forall y.\ \text{invert-order}\ (\text{acomp}\ x\ y) = (\text{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 ⇒ 'a ⇒ bool where
ptrans-comp acomp x ⟷ (\( \forall y z. \) 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 ⇒ bool where
eq-comp acomp ⟷ (\( \forall x. \) peq-comp acomp x)

lemma eq-compD2: eq-comp acomp ⟹ peq-comp acomp x
⟨proof⟩

lemma eq-compI2: (\( \forall x. \) peq-comp acomp x) ⟹ eq-comp acomp
⟨proof⟩

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

lemma trans-compD2: trans-comp acomp ⟹ ptrans-comp acomp x
⟨proof⟩

lemma trans-compI2: (\( \forall x. \) ptrans-comp acomp x) ⟹ trans-comp acomp
⟨proof⟩

definition sym-comp :: 'a comparator ⇒ bool where
sym-comp acomp ⟷ (\( \forall x. \) psym-comp acomp x)
lemma **sym-compD2**:  
\(\text{sym-comp } \text{acomp} \implies \text{psym-comp } \text{acomp } x\)  
⟨proof⟩

lemma **sym-compI2**: \((\forall x. \text{psym-comp } \text{acomp } x) \implies \text{sym-comp } \text{acomp}\)  
⟨proof⟩

lemma **eq-compD**: \(\text{eq-comp } \text{acomp} \implies \text{acomp } x y = \text{Eq } \leftrightarrow x = y\)  
⟨proof⟩

lemma **eq-compI**: \((\forall x y. \text{acomp } x y = \text{Eq } \leftrightarrow x = y) \implies \text{eq-comp } \text{acomp}\)  
⟨proof⟩

lemma **trans-compD**: \(\text{trans-comp } \text{acomp} \implies \text{trans-order } (\text{acomp } x y) (\text{acomp } y z) (\text{acomp } x z)\)  
⟨proof⟩

lemma **trans-compI**: \((\forall x y z. \text{trans-order } (\text{acomp } x y) (\text{acomp } y z) (\text{acomp } x z)) \implies \text{trans-comp } \text{acomp}\)  
⟨proof⟩

lemma **sym-compD**:  
\(\text{sym-comp } \text{acomp} \implies \text{invert-order } (\text{acomp } x y) = (\text{acomp } y x)\)  
⟨proof⟩

lemma **sym-compI**: \((\forall x y. \text{invert-order } (\text{acomp } x y) = (\text{acomp } y x)) \implies \text{sym-comp } \text{acomp}\)  
⟨proof⟩

lemma **eq-sym-trans-imp-comparator**:  
- **assumes** \(\text{eq-comp } \text{acomp}\)  
- **assumes** \(\text{sym-comp } \text{acomp}\)  
- **assumes** \(\text{trans-comp } \text{acomp}\)  
- **shows** \(\text{comparator } \text{acomp}\)  
⟨proof⟩

lemma **comparator-imp-eq-sym-trans**:  
- **assumes** \(\text{comparator } \text{acomp}\)  
- **shows** \(\text{eq-comp } \text{acomp} \text{ sym-comp } \text{acomp} \text{ trans-comp } \text{acomp}\)  
⟨proof⟩

c **context**  
- **fixes** \(\text{acomp} :: \text{a comparator}\)  
- **assumes** \(c :: \text{comparator } \text{acomp}\)  

**begin**  
lemma **comp-to-psym-comp**: \(\text{psym-comp } \text{acomp } x\)  
⟨proof⟩

lemma **comp-to-peq-comp**: \(\text{peq-comp } \text{acomp } x\)  
⟨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 \textit{bool} and \textit{unit}, but implement more efficient ones.

\begin{verbatim}
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⟩
\end{verbatim}

\begin{verbatim}
⟨ML⟩
derive compare bool unit
\end{verbatim}

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

\begin{verbatim}
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
\end{verbatim}

4.9 Defining Compare-Order-Instances for Common Types

\begin{verbatim}
theory Compare-Order-Instances
imports
  Compare-Instances
  ~~/src/HOL/Library/List-lexord
  ~~/src/HOL/Library/Product-Lexorder
  ~~/src/HOL/Library/Option-ord
begin

We now also instantiate class \textit{compare-order} and not only \textit{compare}. Here, we also prove that our definitions do not clash with existing orders on \textit{list},
\end{verbatim}
option, and prod.
For sum we just define the linear orders via their comparator.
derive compare-order sum

instance list :: (compare-order)compare-order
⟨proof⟩

instance prod :: (compare-order, compare-order)compare-order
⟨proof⟩

instance option :: (compare-order)compare-order
⟨proof⟩

end

5 Checking Equality Without ”=”

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

  typedef ('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
\begin{align*}
\text{list-all-eq } [b] &= b \\
\text{list-all-eq } (b1 \# b2 \# bs) &= (b1 \wedge \text{list-all-eq } (b2 \# bs))
\end{align*}

\textit{lemma} list-all-eq: list-all-eq bs \iff (\forall b \in \text{set } bs. b)

\textbf{5.2 Partial Equality Property}

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

\textit{type-synonym} 'a equality = 'a \Rightarrow 'a \Rightarrow \text{bool}

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

\textit{lemma} pequalityD: pequality aeq x \Rightarrow aeq x y \iff x = y

\textit{lemma} pequalityI: (\forall x y. aeq x y \iff x = y) \Rightarrow pequality aeq x

\textbf{5.3 Global equality property}

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

\textit{lemma} equalityD2: equality aeq \Rightarrow pequality aeq x

\textit{lemma} equalityI2: (\forall x. \text{pequality } aeq x) \Rightarrow equality aeq

\textit{lemma} equalityD: equality aeq \Rightarrow aeq x y \iff x = y

\textit{lemma} equalityI: (\forall x y. aeq x y \iff x = y) \Rightarrow equality aeq

\textit{lemma} equality-imp-eq:
equality aeq \Rightarrow aeq = (\text{op } =)

\textit{lemma} eq-equality: equality (\text{op } =)

\textit{lemma} equality-def ': equality f = (f = \text{op } =)

\textit{proof}

20
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. \texttt{int}, \texttt{integer}, \texttt{nibble}, \texttt{nat}, \texttt{char}, \texttt{bool}, \texttt{unit}, \texttt{sum}, \texttt{option}, \texttt{list}, and \texttt{prod}. For types without type parameters, we use plain \texttt{op =}, and for the others we use generated ones. These functions will be essential, when the generator is later on invoked on types, which in their definition use one these types.

derive (eq) equality int integer nibble nat char bool unit
derive equality sum list prod option
end

6 Generating Hash-Functions

theory Hash-Generator
imports
  ../Generator-Aux
  ../Derive-Manager
  ../../Collections/Lib/HashCode
begin
  As usual, in the generator we use a dedicated function to combine the results from evaluating the hash-function of the arguments of a constructor, to deliver the global hash-value.

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

  The first argument of \texttt{hash-combine} originates from evaluating the hash-function on the arguments of a constructor, and the second argument of \texttt{hash-combine} will be static \texttt{magic} numbers which are generated within the generator.
6.1 Improved Code for Non-Lazy Languages

lemma hash-combine-unfold:
  hash-combine [] [x] = x
  hash-combine (y # ys) (z # zs) = y * z + hash-combine ys zs
  ⟨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 \textit{compare-order-instance}.

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

Use \texttt{op =} 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
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
**9.6 Examples from IsaFoR**

```plaintext
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.

```plaintext
datatype (′a, ′b) complex =  
  C1 nat 'a ttree × rat + ('a,'b) complex list  |  
  C2 ('a, 'b) complex list tree tree ('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

