Deriving class instances for datatypes.*

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 (linear) orders or hash-functions which are required in the Isabelle Collection Framework [1] and the Container Framework [2]. Moreover, for the tactic of Huffman and Krauss to show that a datatype is countable, we implemented a wrapper so that this tactic becomes accessible in our framework.

Our formalization was performed as part of the IsaFoR/CeTA project\(^1\) [3]. With our new tactic we could completely remove tedious proofs for linear orders of two datatypes.

Contents

1 Important Information 2

2 Derive Manager 2

3 Generating linear orders for datatypes 3

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3.2 Implementation Notes . . . . . . . . . . . . . . . . . . . . . . 4
3.3 Features and Limitations . . . . . . . . . . . . . . . . . . . . 4
3.4 Installing the generator . . . . . . . . . . . . . . . . . . . . . 4

4 Hash functions 4

4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
4.2 Features and Limitations . . . . . . . . . . . . . . . . . . . . 5
4.3 Installing the generator . . . . . . . . . . . . . . . . . . . . . 5

*Supported by FWF (Austrian Science Fund) project P22767-N13.
\(^1\)http://cl-informatik.uibk.ac.at/software/ceta
1 Important Information

The described generators are outdated as they are based on the old datatype package. Generators for the new datatypes are available in the AFP entry “Deriving”.

2 Derive Manager

decline 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.
3 Generating linear orders for datatypes

The order generator registers itself at the derive-manager for the classes ord, order, and linorder. To be more precise, it automatically generates the two functions \( op \leq \) and \( op < \) for some datatype \( dtype \) and proves the following instantiations.

- instantiation \( dtype :: (ord,\ldots,ord) \) ord
- instantiation \( dtype :: (order,\ldots,order) \) order
- instantiation \( dtype :: (linorder,\ldots,linorder) \) linorder

All the non-recursive types that are used in the datatype must have similar instantiations. For recursive type-dependencies this is automatically generated.

For example, for the datatype \( tree = \text{Leaf nat} \mid \text{Node "tree list"} \) we require that \( nat \) is already in linorder, whereas for \( list \) nothing is required, since for the \( tree \) datatype the \( list \) is only used recursively.

However, if we define \( \text{datatype tree = Leaf "nat list"} \mid \text{Node tree tree} \) then \( list \) must provide the above instantiations.

Note that when calling the generator for linorder, it will automatically also derive the instantiations for order, which in turn invokes the generator for ord. A later invocation of linorder after order or ord is not possible.
3.2 Implementation Notes

The generator uses the recursors from the datatype package to define a lexicographic order. E.g., for a declaration `datatype 'a tree = Empty | Node "'a tree" 'a "'a tree"` this will semantically result in

\[(\text{Empty} < \text{Node } _ _ _) = \text{True} \]
\[(\text{Node } 11 \ 12 \ 13 < \text{Node } r1 \ r2 \ r3) = \]
\[(11 < r1 || 11 = r1 && (12 < r2 || 12 = r2 && 13 < r3)) \]
\[(<_<_>) = \text{False} \]
\[(1 <= r) = (1 < r || 1 = r) \]

The desired properties (like \([x < y; y < z] \Rightarrow x < z\)) of the orders are all proven using induction (with the induction theorem from the datatype on \(x\)), and afterwards there is a case distinction on the remaining variables, i.e., here \(y\) and \(z\). If the constructors of \(x\), \(y\), and \(z\) are different always some basic tactic is invoked. In the other case (identical constructors) for each property a dedicated tactic was designed.

3.3 Features and Limitations

The order generator has been developed mainly for datatypes without explicit mutual recursion. For mutual recursive datatypes—like `datatype a = C b and b = D a a`—only for the first mentioned datatype—here \(a\)—the instantiations of the order-classes are derived.

Indirect recursion like in `datatype tree = Leaf nat | Node "tree list"` should work without problems.

3.4 Installing the generator

\[\text{lemma linear-cases: } (x :: 'a :: linorder) = y \lor x < y \lor y < x \text{ by auto} \]

ML-file `order-generator.ML`

end

4 Hash functions

\[\text{theory Hash-Generator} \]
\[\text{imports} \]
\[../Collections/Lib/HashCode \]
\[Derive-Aux \]
\[\text{begin} \]

4.1 Introduction

The interface for hash-functions is defined in the class `hashable` which has been developed as part of the Isabelle Collection Framework [1]. It requires
a hash-function (\textit{hashcode}), a bounded hash-function (\textit{bounded-hashcode}), and a default hash-table size (\textit{def-hashmap-size}).

The \textit{hashcode} function for each datatype are created by instantiating the recursors of that datatype appropriately. E.g., for \textbf{datatype} \texttt{'}a test = C1 \texttt{'}a \ 'a | C2 \texttt{"'}a test list"\texttt{"} we get a hash-function which is equivalent to

\begin{align*}
\text{hashcode (C1 a b)} &= c1 \ast \text{hashcode a} + c2 \ast \text{hashcode b} \\
\text{hashcode (C2 Nil)} &= c3 \\
\text{hashcode (C2 (a # as))} &= c4 \ast \text{hashcode a} + c5 \ast \text{hashcode as}
\end{align*}

where each $c_i$ is a non-negative 32-bit number which is dependent on the datatype name, the constructor name, and the occurrence of the argument (i.e., in the example $c1$ and $c2$ will usually be different numbers.) These parameters are used in linear combination with prime numbers to hopefully get some useful hash-function.

The \textit{bounded-hashcode} functions are constructed in the same way, except that after each arithmetic operation a modulo operation is performed.

Finally, the default hash-table size is just set to 10, following Java’s default hash-table constructor.

\section{Features and Limitations}

We get same limitation as for the order generator. For mutual recursive datatypes, only for the first mentioned datatype the instantiations of the \textit{hashable}-class are derived.

\section{Installing the generator}

\textbf{lemma} hash-mod-lemma: $1 < (n :: \text{nat}) \implies x \mod n < n$ \textbf{by auto}

\textbf{ML-file} hash-generator.ML

end

\section{Countable Datatypes}

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

end
5.1 Installing the tactic

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

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

6 Loading derive-commands

```ml
theory Derive
imports
  Order-Generator
  Hash-Generator
  ../Deriving/Countable-Generator/Countable-Generator
begin
  We just load the commands to derive (linear) orders, 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 of lightweight containers and Show.
  print-derives
end
```

7 Examples

```ml
theory Derive-Examples
imports
  Derive
```
Rat
begin

7.1 Register standard existing types
derive linorder list sum prod

7.2 Without nested recursion
datatype 'a bintree = BEmpty | BNode 'a bintree 'a 'a bintree
derive linorder bintree
derive hashable bintree
derive countable bintree

7.3 Using other datatypes
datatype nat-list-list = NNil | CCons nat list nat-list-list
derive linorder nat-list-list
derive hashable nat-list-list
derive countable nat-list-list

7.4 Explicit mutual recursion
datatype 'a mtree = MEmpty | MNode 'a 'a mtree-list and
'a mtree-list = MNil | MCons 'a mtree 'a mtree-list
derive linorder mtree
derive hashable mtree
derive countable mtree

7.5 Implicit mutual recursion
datatype 'a tree = Empty | Node 'a 'a tree list
datatype-compat tree
derive linorder tree
derive hashable tree
derive countable tree
datatype 'a ttree = TEmpty | TNode 'a 'a ttree list tree
datatype-compat ttree
derive linorder ttree
derive hashable ttree
derive countable ttree
7.6 Examples from IsaFoR

datatype (′f,′v) term = Var ′v | Fun ′f (′f,′v) term list

datatype-compat term

datatype (′f, ′l) lab =
   Lab (′f, ′l) lab ′l
| FunLab (′f, ′l) lab (′f, ′l) lab list
| UnLab ′f
| Sharp (′f, ′l) lab

datatype-compat lab

derive linorder term lab
derive countable term lab
derive hashable term lab

7.7 A complex datatype

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

datatype (′a, ′b) complex =
   C1 nat ′a ttree |
   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

datatype-compat complex complex2

derive linorder complex
derive hashable complex
derive countable complex

datatype-compat complex complex2

derive linorder complex
derive hashable complex
derive countable complex

don't have a

8 Acknowledgements

We thank

- Lukas Bulwahn and Brian Huffman for the discussion on a generic
derive command and the pointer to the tactic for countability.

- Alexander Krauss for pointing me to the recursors of the datatype
package.

- Peter Lammich for the inspiration of developing a hash-function gen-
erator.
• Andreas Lochbihler for the inspiration of developing generators for the container framework.

• Christian Urban for his cookbook about the ML-level of Isabelle.

• Stefan Berghofer, Cezary Kaliszyk, and Tobias Nipkow for their explanations on several Isabelle related questions.

References

