Deriving class instances for datatypes.*

René Thiemann

December 14, 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 (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

*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The described generators are outdated as they are based on the old datatype package.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".

theory *Derive-Aux*
imports
 Deriving.Derive-Manager
begin

ML-file ‹*derive-aux.ML*›

end

2 Generating linear orders for datatypes

theory *Order-Generator*
imports
 Derive-Aux
begin

2.1 Introduction

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 $(\leq)$ and $(<)$ for some datatype `dtype` and proves the following instantiations.

- `instantiation dtype :: (ord,...,ord) ord`
- `instantiation dtype :: (order,...,order) order`
- `instantiation dtype :: (linorder,...,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 = Leaf nat | 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 `datatype tree = Leaf "nat list" | 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 invokation of *linorder* after *order* or *ord* is not possible.

2.2 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

```
(Empty < Node _ _ _) = True
(Node l1 l2 l3 < Node r1 r2 r3) =
  (l1 < r1 || l1 = r1 && (l2 < r2 || l2 = r2 && l3 < r3))
(_ < _) = False
(l <= r) = (l < r || l = r)
```

The desired properties (like $[\![x < y;\ y < z]\!] \Longrightarrow x < z$) 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.

2.3 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.

2.4 Installing the generator

lemma *linear-cases*: $(x :: 'a :: linorder) = y \lor x < y \lor y < x$ **by** *auto*

ML-file ‹*order-generator.ML*›

end

3 Hash functions

theory *Hash-Generator*
imports
Collections.HashCode
Derive-Aux
begin

3.1 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 (*hashcode*), a bounded hash-function (*bounded-hashcode*), and a default hash-table size (*def-hashmap-size*).

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

```
hashcode (C1 a b) = c1 * hashcode a + c2 * hashcode b
hashcode (C2 Nil) = c3
hashcode (C2 (a # as)) = c4 * hashcode a + c5 * hashcode as
```

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 *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.

3.2 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 *hashable*-class are derived.

3.3 Installing the generator

lemma *hash-mod-lemma*: $1 < (n :: nat) \Longrightarrow x \ mod \ n < n$ **by** *auto*

ML-file ‹*hash-generator.ML*›

end

4 Loading derive-commands

theory *Derive*
imports
  Order-Generator
  Hash-Generator
  Deriving.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

5 Examples

theory *Derive-Examples*
imports
  Derive
  HOL.Rat
begin

5.1 Register standard existing types

derive *linorder list sum prod*

5.2 Without nested recursion

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

derive *linorder bintree*
derive *hashable bintree*
derive *countable bintree*

5.3 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*

5.4 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*

5.5 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*

5.6 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*

5.7 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

end
```

6 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 generator.
- 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

[1] P. Lammich and A. Lochbihler. The Isabelle collections framework. In *Proc. ITP'10*, volume 6172 of *LNCS*, pages 339–354, 2010.

[2] A. Lochbihler. Light-weight containers for isabelle: Efficient, extensible, nestable. In *Proc. ITP'13*, volume 7998 of *LNCS*, pages 116–132, 2013.

[3] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.