

# Deriving generic class instances for datatypes

Jonas Rädle and Lars Hupel

August 7, 2022

## Abstract

We provide a framework for automatically deriving instances for generic type classes. Our approach is inspired by Haskell’s *generic-deriving* package [1] and Scala’s *shapeless* library [2].

In addition to generating the code for type class functions, we also attempt to automatically prove type class laws for these instances. As of now, however, some manual proofs are still required for recursive datatypes.

## Contents

<b>1</b>	<b>Tagged Sum-of-Products Representation</b>	<b>1</b>
<b>2</b>	<b>Derive</b>	<b>3</b>
<b>3</b>	<b>Examples</b>	<b>3</b>
3.1	Example Datatypes . . . . .	3
3.2	Equality . . . . .	4
3.3	Encoding . . . . .	6
3.4	Algebraic Classes . . . . .	8
3.5	Show . . . . .	11
3.6	Classes with Laws . . . . .	12
3.6.1	Equality . . . . .	12
3.6.2	Algebraic Classes . . . . .	17

## 1 Tagged Sum-of-Products Representation

This theory sets up a version of the sum-of-products representation that includes constructor and selector names. For an example of a type class that uses this representation see `Derive_Show`.

```
theory Tagged-Prod-Sum  
imports Main  
begin
```

**context begin**

**qualified datatype** ('a, 'b) *prod* = *Prod string option string option 'a 'b*  
**qualified datatype** ('a, 'b) *sum* = *Inl string option 'a | Inr string option 'b*

**qualified definition** *fst* **where** *fst* *p* = (*case p of (Prod - - a -) ⇒ a*)  
**qualified definition** *snd* **where** *snd* *p* = (*case p of (Prod - - - b) ⇒ b*)  
**qualified definition** *sel-name-fst* **where** *sel-name-fst* *p* = (*case p of (Prod s - -) ⇒ s*)  
**qualified definition** *sel-name-snd* **where** *sel-name-snd* *p* = (*case p of (Prod - s -) ⇒ s*)

**qualified definition** *constr-name* **where** *constr-name* *x* = (*case x of (Inl s -) ⇒ s | (Inr s -) ⇒ s*)

**end**

**lemma** *measure-tagged-fst[measure-function]*: *is-measure f ⇒ is-measure*  
*(λ p. f (Tagged-Prod-Sum.fst p))*  
**by** (*rule is-measure-trivial*)

**lemma** *measure-tagged-snd[measure-function]*: *is-measure f ⇒ is-measure*  
*(λ p. f (Tagged-Prod-Sum.snd p))*  
**by** (*rule is-measure-trivial*)

**lemma** *size-tagged-prod-simp*:  
*Tagged-Prod-Sum.prod.size-prod f g p = f (Tagged-Prod-Sum.fst p) + g*  
*(Tagged-Prod-Sum.snd p) + Suc 0*  
**apply** (*induct p*)  
**by** (*simp add: Tagged-Prod-Sum.fst-def Tagged-Prod-Sum.snd-def*)

**lemma** *size-tagged-sum-simp*:  
*Tagged-Prod-Sum.sum.size-sum f g x = (case x of Tagged-Prod-Sum.Inl -*  
*a ⇒ f a + Suc 0 | Tagged-Prod-Sum.Inr - b ⇒ g b + Suc 0)*  
**apply** (*induct x*)  
**by** *auto*

**lemma** *size-tagged-prod-measure*:  
*is-measure f ⇒ is-measure g ⇒ is-measure (Tagged-Prod-Sum.prod.size-prod*  
*f g)*  
**by** (*rule is-measure-trivial*)

```

lemma size-tagged-sum-measure:
  is-measure f  $\implies$  is-measure g  $\implies$  is-measure (Tagged-Prod-Sum.sum.size-sum
f g)
by (rule is-measure-trivial)

end

```

## 2 Derive

This theory includes the Isabelle/ML code needed for the derivation and exports the two keywords `derive_generic` and `derive_generic_setup`.

```

theory Derive
  imports Main Tagged-Prod-Sum
  keywords derive-generic derive-generic-setup :: thy-goal
begin

context begin

qualified definition iso :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('b  $\Rightarrow$  'a)  $\Rightarrow$  bool where
iso from to = (( $\forall$  a. to (from a) = a)  $\wedge$  ( $\forall$  b . from (to b) = b))

lemma iso-intro: ( $\wedge$  a. to (from a) = a)  $\implies$  ( $\wedge$  b. from (to b) = b)  $\implies$  iso
from to
  unfolding iso-def by simp

end

ML-file  $\langle$ derive-util.ML $\rangle$ 
ML-file  $\langle$ derive-laws.ML $\rangle$ 
ML-file  $\langle$ derive-setup.ML $\rangle$ 
ML-file  $\langle$ derive.ML $\rangle$ 

end

```

## 3 Examples

### 3.1 Example Datatypes

```

theory Derive-Datatypes
imports Main
begin

```

```
datatype simple = A (num: nat) | B (left:nat) (right:nat) | C
```

```
datatype ('a,'b) either = L 'a | R 'b
```

```
datatype 'a tree = Leaf | Node 'a 'a tree 'a tree
```

```
datatype even-nat = Even-Zero | Even-Succ odd-nat  
and odd-nat = Odd-Succ even-nat
```

```
datatype ('a,'b) exp = Term ('a,'b) trm | Sum (left:('a,'b) trm) (right:('a,'b)  
exp)  
and ('a,'b) trm = Factor ('a,'b) fct | Prod ('a,'b) fct ('a,'b) trm  
and ('a,'b) fct = Const 'a | Var (v:'b) | Expr ('a,'b) exp
```

```
end
```

## 3.2 Equality

```
theory Derive-Eq  
  imports Main ../Derive Derive-Datatypes  
begin
```

```
class eq =  
  fixes eq :: 'a ⇒ 'a ⇒ bool
```

```
instantiation nat and unit:: eq
```

```
begin
```

```
  definition eq-nat : eq (x::nat) y ⟷ x = y
```

```
  definition eq-unit-def: eq (x::unit) y ⟷ True
```

```
  instance ..
```

```
end
```

```
instantiation prod and sum :: (eq, eq) eq
```

```
begin
```

```
  definition eq-prod-def: eq x y ⟷ (eq (fst x) (fst y)) ∧ (eq (snd x) (snd  
y))
```

```
  definition eq-sum-def: eq x y = (case x of Inl a ⇒ (case y of Inl b ⇒ eq  
a b | Inr b ⇒ False)
```

```
    | Inr a ⇒ (case y of Inl b ⇒ False | Inr
```

$b \Rightarrow eq\ a\ b))$

**instance ..**  
**end**

**derive-generic** *eq simple* .

**lemma** *eq (A 4) (A 4) by eval*  
**lemma** *eq (A 6) (A 4)  $\longleftrightarrow$  False by eval*  
**lemma** *eq C C by eval*  
**lemma** *eq (B 4 5) (B 4 5) by eval*  
**lemma** *eq (B 4 4) (A 3)  $\longleftrightarrow$  False by eval*  
**lemma** *eq C (A 4)  $\longleftrightarrow$  False by eval*

**derive-generic** *eq either* .

**lemma** *eq (L (3::nat)) (R 3)  $\longleftrightarrow$  False by code-simp*  
**lemma** *eq (L (3::nat)) (L 3) by code-simp*  
**lemma** *eq (L (3::nat)) (L 4)  $\longleftrightarrow$  False by code-simp*

**derive-generic** *eq list* .

**lemma** *eq ([]::(nat list)) [] by eval*  
**lemma** *eq ([1,2,3]::(nat list)) [1,2,3] by eval*  
**lemma** *eq [(1::nat)] [1,2]  $\longleftrightarrow$  False by eval*

**derive-generic** *eq tree* .

**lemma** *eq Leaf Leaf by code-simp*  
**lemma** *eq (Node (1::nat) Leaf Leaf) Leaf  $\longleftrightarrow$  False by eval*  
**lemma** *eq (Node (1::nat) Leaf Leaf) (Node (1::nat) Leaf Leaf) by eval*  
**lemma** *eq (Node (1::nat) (Node 2 Leaf Leaf) (Node 3 Leaf Leaf)) (Node (1::nat) (Node 2 Leaf Leaf) (Node 4 Leaf Leaf))  $\longleftrightarrow$  False by eval*

**derive-generic** *eq even-nat* .  
**derive-generic** *eq exp* .

**lemma** *eq Even-Zero Even-Zero* **by** *eval*  
**lemma** *eq Even-Zero (Even-Succ (Odd-Succ Even-Zero))*  $\longleftrightarrow$  *False* **by** *eval*  
**lemma** *eq (Odd-Succ (Even-Succ (Odd-Succ Even-Zero))) (Odd-Succ (Even-Succ (Odd-Succ Even-Zero)))* **by** *eval*  
**lemma** *eq (Odd-Succ (Even-Succ (Odd-Succ Even-Zero))) (Odd-Succ (Even-Succ (Odd-Succ (Even-Succ (Odd-Succ (Even-Succ (Odd-Succ Even-Zero)))))))*  
 $\longleftrightarrow$  *False* **by** *eval*

**lemma** *eq (Const (1::nat)) (Const (1::nat))* **by** *code-simp*  
**lemma** *eq (Const (1::nat)) (Var (1::nat))*  $\longleftrightarrow$  *False* **by** *eval*  
**lemma** *eq (Term (Prod (Const (1::nat)) (Factor (Const (2::nat))))) (Term (Prod (Const (1::nat)) (Factor (Const (2::nat)))))*  
**by** *code-simp*  
**lemma** *eq (Term (Prod (Const (1::nat)) (Factor (Const (2::nat))))) (Term (Prod (Const (1::nat)) (Factor (Const (3::nat)))))*  
 $\longleftrightarrow$  *False* **by** *code-simp*

**end**

### 3.3 Encoding

**theory** *Derive-Encode*  
**imports** *Main ../Derive Derive-Datatypes*  
**begin**

**class** *encodeable* =  
**fixes** *encode* :: 'a  $\Rightarrow$  *bool list*

**instantiation** *nat and unit:: encodeable*

**begin**  
**fun** *encode-nat* :: *nat*  $\Rightarrow$  *bool list* **where**  
*encode-nat* 0 = [] |  
*encode-nat* (Suc n) = *True* # (*encode* n)

**definition** *encode-unit*: *encode* (*x::unit*) = []

**instance** ..

**end**

**instantiation** *prod and sum* :: (*encodeable*, *encodeable*) *encodeable*  
**begin**

**definition** *encode-prod-def*:  $encode\ x = append\ (encode\ (fst\ x))\ (encode\ (snd\ x))$

**definition** *encode-sum-def*:  $encode\ x = (case\ x\ of\ Inl\ a \Rightarrow False\ \# \ encode\ a$   
 $| \ Inr\ a \Rightarrow True\ \# \ encode\ a)$

**instance** ..  
**end**

**derive-generic** *encodeable simple* .  
**derive-generic** *encodeable either* .

**lemma**  $encode\ (B\ 3\ 4) = [True, False, True, True, True, True, True, True, True, True]$  **by** *eval*

**lemma**  $encode\ C = [True, True]$  **by** *eval*

**lemma**  $encode\ (R\ (3::nat)) = [True, True, True, True]$  **by** *code-simp*

**derive-generic** *encodeable list* .  
**derive-generic** *encodeable tree* .

**lemma**  $encode\ [1,2,3,4::nat]$

$= [True, True, True, True, True, True, True, True, True, True, True, True, True, True, False]$  **by** *eval*

**lemma**  $encode\ (Node\ (3::nat)\ (Node\ 1\ Leaf\ Leaf)\ (Node\ 2\ Leaf\ Leaf))$

$= [True, True, True, True, True, True, False, False, True, True, True, False, False]$  **by** *eval*

**derive-generic** *encodeable even-nat* .  
**derive-generic** *encodeable exp* .

**lemma**  $encode\ (Odd-Succ\ (Even-Succ\ (Odd-Succ\ Even-Zero)))$

$= [True, False, True, True, False, False]$  **by** *eval*

**lemma**  $encode\ (Term\ (Prod\ (Const\ (1::nat))\ (Factor\ (Const\ (2::nat))))$

$= [False, False, True, False, True, True, True, False, True, True, False, False, True, True, False, True, True]$

**by** *code-simp*

**end**

### 3.4 Algebraic Classes

```
theory Derive-Algebra
imports Main ../Derive Derive-Datatypes
begin

class semigroup =
  fixes mult :: 'a ⇒ 'a ⇒ 'a (infixl  $\otimes$  70)

class monoidl = semigroup +
  fixes neutral :: 'a (1)

class group = monoidl +
  fixes inverse :: 'a ⇒ 'a

instantiation nat and unit:: semigroup
begin
  definition mult-nat : mult (x::nat) y = x + y
  definition mult-unit-def: mult (x::unit) y = x
  instance ..
end
instantiation nat and unit:: monoidl
begin
  definition neutral-nat : neutral = (0::nat)
  definition neutral-unit-def: neutral = ()
  instance ..
end
instantiation nat and unit:: group
begin
  definition inverse-nat : inverse (i::nat) = 1 - i
  definition inverse-unit-def: inverse u = ()
  instance ..
end

instantiation prod and sum :: (semigroup, semigroup) semigroup
begin
  definition mult-prod-def: x  $\otimes$  y = (fst x  $\otimes$  fst y, snd x  $\otimes$  snd y)
  definition mult-sum-def: x  $\otimes$  y = (case x of Inl a ⇒ (case y of Inl b ⇒
Inl (a  $\otimes$  b) | Inr b ⇒ Inl a)
```



|  $Inr\ a \Rightarrow (case\ y\ of\ Inl\ b \Rightarrow Inr\ a \mid Inr$

$b \Rightarrow Inr\ (a \otimes b))$ )

**instance ..**

**end**

**instantiation** *prod and sum :: (monoidl, monoidl) monoidl*

**begin**

**definition** *neutral-prod-def: neutral = (neutral, neutral)*

**definition** *neutral-sum-def: neutral = Inl neutral*

**instance ..**

**end**

**instantiation** *prod and sum :: (group, group) group*

**begin**

**definition** *inverse-prod-def: inverse p = (inverse (fst p), inverse (snd p))*

**definition** *inverse-sum-def: inverse x = (case x of Inl a => (Inl (inverse a))*

|  $Inr\ b \Rightarrow Inr\ (inverse\ b)$ )

**instance ..**

**end**

**derive-generic** *semigroup simple .*

**derive-generic** *monoidl simple .*

**derive-generic** *group simple .*

**lemma**  $(B\ 1\ 6) \otimes (B\ 4\ 5) = B\ 4\ 11$  **by** *eval*

**lemma**  $(A\ 2) \otimes (A\ 3) = A\ 5$  **by** *eval*

**lemma**  $(B\ 1\ 6) \otimes 1 = B\ 0\ 6$  **by** *eval*

**derive-generic** *group either .*

**lemma**  $(L\ 3) \otimes ((L\ 4)::(nat, nat)\ either) = L\ 7$  **by** *eval*

**lemma**  $(R\ (2::nat)) \otimes (L\ (3::nat)) = R\ 2$  **by** *eval*

**derive-generic** *semigroup list .*

**derive-generic** *monoidl list .*

**derive-generic** *group list .*

**derive-generic** *semigroup tree .*

**derive-generic** *monoidl tree* .  
**derive-generic** *group tree* .

**lemma**  $[1,2,3,4::nat] \otimes [1,2,3] = [2,4,6,4]$  **by** *eval*  
**lemma** *inverse*  $[1,2,3::nat] = [0,0,0]$  **by** *eval*

**derive-generic** *semigroup even-nat* .  
**derive-generic** *monoidl even-nat* .  
**derive-generic** *group even-nat* .  
**derive-generic** *semigroup exp* .

**instantiation** *exp and trm and fct* :: (*monoidl,monoidl*) *monoidl*  
**begin**  
  **definition** *neutral-fct* **where** *neutral-fct* = *Const neutral*  
  **definition** *neutral-trm* **where** *neutral-trm* = *Factor neutral*  
  **definition** *neutral-exp* **where** *neutral-exp* = *Term neutral*  
  **instance** ..  
**end**

**setup** <  
  (*Derive.add-inst-info class* <*monoidl*> **type-name** <*fct*> [*@{thm neutral-fct-def}*])  
  #>  
  (*Derive.add-inst-info class* <*monoidl*> **type-name** <*trm*> [*@{thm neutral-trm-def}*])  
  #>  
  (*Derive.add-inst-info class* <*monoidl*> **type-name** <*exp*> [*@{thm neutral-exp-def}*])  
  >

**derive-generic** *group exp* .

**lemma** (*Odd-Succ (Even-Succ (Odd-Succ Even-Zero))*)  $\otimes$  (*Odd-Succ Even-Zero*)

  = *Odd-Succ (Even-Succ (Odd-Succ Even-Zero))* **by** *eval*

**lemma** *inverse (Odd-Succ Even-Zero) = Odd-Succ Even-Zero* **by** *eval*

**lemma** (*Term (Prod ((Const 1)::(nat, nat) fct) (Factor (Const (2::nat))))*)

$\otimes$  (*Term (Prod (Const (2::nat)) (Factor ((Const 2)::(nat, nat) fct)))*)

  = *Term (Prod (Const 3) (Factor (Const 4)))* **by** *eval*

**end**

### 3.5 Show

```
theory Derive-Show
imports Main ../Derive Derive-Datatypes
begin

class showable =
  fixes print :: 'a ⇒ string

fun string-of-nat :: nat ⇒ string
where
  string-of-nat n = (if n < 10 then [(char-of :: nat ⇒ char) (48 + n)] else
    string-of-nat (n div 10) @ [(char-of :: nat ⇒ char) (48 + (n mod 10))])

instantiation nat and unit:: showable
begin
  definition print-nat: print (n::nat) = string-of-nat n
  definition print-unit: print (x::unit) = ""
  instance ..
end

instantiation Tagged-Prod-Sum.prod and Tagged-Prod-Sum.sum :: (showable,
showable) showable
begin
definition print-prod-def:
  print (x::('a,'b) Tagged-Prod-Sum.prod) =
    (case Tagged-Prod-Sum.sel-name-fst x of
      None ⇒ (print (Tagged-Prod-Sum.fst x))
      | Some s ⇒ "(" @ s @ ": " @ (print (Tagged-Prod-Sum.fst x)) @ ")")
    @
    " "
    @
    (case Tagged-Prod-Sum.sel-name-snd x of
      None ⇒ (print (Tagged-Prod-Sum.snd x))
      | Some s ⇒ "(" @ s @ ": " @ (print (Tagged-Prod-Sum.snd x)) @ ")")

definition print-sum-def: print (x::('a,'b) Tagged-Prod-Sum.sum) =
  (case x of (Tagged-Prod-Sum.Inl s a) ⇒ (case s of None ⇒ print a | Some
c ⇒ "(" @ c @ " " @ (print a) @ ")")
    | (Tagged-Prod-Sum.Inr s b) ⇒ (case s of None ⇒ print b | Some
c ⇒ "(" @ c @ " " @ (print b) @ ")")
  instance ..
end
```

```

declare [[ML-print-depth=30]]

derive-generic (metadata) showable simple .
derive-generic (metadata) showable either .

value print (A 3)
value print (B 1 2)
value [simp] print (L (2::nat))
value print C

derive-generic (metadata) showable list .
derive-generic (metadata) showable tree .

value print [1,2::nat]
value print (Node (3::nat) (Node 1 Leaf Leaf) (Node 2 Leaf Leaf))

derive-generic (metadata) showable even-nat .
derive-generic (metadata) showable exp .

value print (Odd-Succ (Even-Succ (Odd-Succ Even-Zero)))
value [simp] print (Sum (Factor (Const (0::nat))) (Term (Prod (Const (1::nat)) (Factor (Const (2::nat)))))))

end

3.6 Classes with Laws

3.6.1 Equality

theory Derive-Eq-Laws
  imports Main ../Derive Derive-Datatypes
begin

class eq =
  fixes eq :: 'a ⇒ 'a ⇒ bool
  assumes refl: eq x x and
           sym: eq x y ⇒ eq y x and

```

*trans*:  $eq\ x\ y \implies eq\ y\ z \implies eq\ x\ z$

**derive-generic-setup** *eq*  
  **unfolding** *eq-class-law-def*  
  **by** *blast*

**lemma** *eq-law-eq*: *eq-class-law eq*  
  **unfolding** *eq-class-law-def*  
  **using** *eq-class.axioms* **unfolding** *class.eq-def* .

**instantiation** *nat* and *unit* :: *eq*

**begin**

**definition** *eq-nat-def* :  $eq\ (x::nat)\ y \longleftrightarrow x = y$

**definition** *eq-unit-def*:  $eq\ (x::unit)\ y \longleftrightarrow True$

**instance proof**

**fix** *x y z* :: *nat*

**show**  $eq\ x\ x$  **unfolding** *eq-nat-def* **by** *simp*

**show**  $eq\ x\ y \implies eq\ y\ x$  **unfolding** *eq-nat-def* **by** *simp*

**show**  $eq\ x\ y \implies eq\ y\ z \implies eq\ x\ z$  **unfolding** *eq-nat-def* **by** *simp*

**next**

**fix** *x y z* :: *unit*

**show**  $eq\ x\ x$  **unfolding** *eq-unit-def* **by** *simp*

**show**  $eq\ x\ y \implies eq\ y\ x$  **unfolding** *eq-unit-def* **by** *simp*

**show**  $eq\ x\ y \implies eq\ y\ z \implies eq\ x\ z$  **unfolding** *eq-unit-def* **by** *simp*

**qed**

**end**

**instantiation** *prod* and *sum* :: (*eq*, *eq*) *eq*

**begin**

**definition** *eq-prod-def*:  $eq\ x\ y \longleftrightarrow (eq\ (fst\ x)\ (fst\ y)) \wedge (eq\ (snd\ x)\ (snd\ y))$

**definition** *eq-sum-def*:  $eq\ x\ y = (case\ x\ of\ Inl\ a \Rightarrow (case\ y\ of\ Inl\ b \Rightarrow eq\ a\ b \mid Inr\ b \Rightarrow False) \mid Inr\ a \Rightarrow (case\ y\ of\ Inl\ b \Rightarrow False \mid Inr\ b \Rightarrow eq\ a\ b))$

**instance proof**

**fix** *x y z* :: (*'a*::*eq*)  $\times$  (*'b*::*eq*)

**show**  $eq\ x\ x$  **unfolding** *eq-prod-def* **by** (*simp add: eq-class.refl*)

**show**  $eq\ x\ y \implies eq\ y\ x$  **unfolding** *eq-prod-def* **by** (*simp add: eq-class.sym*)

**show**  $eq\ x\ y \implies eq\ y\ z \implies eq\ x\ z$  **unfolding** *eq-prod-def* **by** (*meson eq-class.trans*)

**next**

**fix** *x y z* :: (*'a*::*eq*)  $+$  (*'b*::*eq*)

```

show eq x x unfolding eq-sum-def by (simp add: sum.case-eq-if eq-class.refl)
  show eq x y  $\implies$  eq y x unfolding eq-sum-def by (metis eq-class.sym
sum.case-eq-if)
  show eq x y  $\implies$  eq y z  $\implies$  eq x z
    unfolding eq-sum-def
    apply (simp only: sum.case-eq-if)
    apply (cases isl x; cases isl y; cases isl z)
    by (auto simp add: eq-class.trans)
qed
end

```

**derive-generic** eq simple .

```

lemma eq (A 4) (A 4) by eval
lemma eq (A 6) (A 4)  $\longleftrightarrow$  False by eval
lemma eq C C by eval
lemma eq (B 4 5) (B 4 5) by eval
lemma eq (B 4 4) (A 3)  $\longleftrightarrow$  False by eval
lemma eq C (A 4)  $\longleftrightarrow$  False by eval

```

**derive-generic** eq either .

```

lemma eq (L (3::nat)) (R 3)  $\longleftrightarrow$  False by code-simp
lemma eq (L (3::nat)) (L 3) by code-simp
lemma eq (L (3::nat)) (L 4)  $\longleftrightarrow$  False by code-simp

```

**derive-generic** eq list

```

proof goal-cases
  case (1 x)
  then show ?case
  proof (induction x)
    case (In y)
    then show ?case
      apply(cases y)
      by (auto simp add: Derive-Eq-Laws.eq-mulistF.simps eq-unit-def eq-class.refl)
  qed
next
  case (2 x y)
  then show ?case

```

```

proof (induction y arbitrary: x)
  case (In y)
  then show ?case
    apply(cases x; cases y; hypsubst-thin)
    apply (simp add: Derive-Eq-Laws.eq-mulistF.simps sum.case-eq-if
eq-unit-def)
    apply(metis old.sum.simps(5))
    unfolding sum-set-defs prod-set-defs
    apply (simp add: Derive-Eq-Laws.eq-mulistF.simps sum.case-eq-if)
    using eq-class.sym by fastforce
  qed
next
  case ( $\exists$  x y z)
  then show ?case
  proof (induction x arbitrary: y z)
    case (In x')
    then show ?case
      apply(cases x')
      apply (cases y; cases z; hypsubst-thin)
      apply (simp add: Derive-Eq-Laws.eq-mulistF.simps sum.case-eq-if
eq-unit-def)
      apply (metis sum.case-eq-if)
      apply(cases y; cases z; hypsubst-thin)
      unfolding sum-set-defs prod-set-defs
      apply (simp add: Derive-Eq-Laws.eq-mulistF.simps eq-unit-def snds.intros)
      apply (simp only: sum.case-eq-if)
      by (meson eq-class.trans)
    qed
  qed

```

```

lemma eq ([]::(nat list)) [] by eval
lemma eq ([1,2,3]::(nat list)) [1,2,3] by eval
lemma eq [(1::nat)] [1,2]  $\longleftrightarrow$  False by eval

```

```

derive-generic eq tree
proof goal-cases
  case (1 x)
  then show ?case
  proof (induction x)
    case (In y)
    then show ?case
      apply(cases y)
      by (auto simp add: Derive-Eq-Laws.eq-mutreeF.simps eq-unit-def
eq-class.refl)
  qed

```

```

qed
next
  case (2 x y)
  then show ?case
  proof (induction y arbitrary: x)
    case (In y)
    then show ?case
      apply(cases x; cases y; hypsubst-thin)
      apply (simp add: Derive-Eq-Laws.eq-mutreeF.simps sum.case-eq-if
eq-unit-def)
      apply(metis old.sum.simps(5))
      unfolding sum-set-defs prod-set-defs
      apply (simp add: Derive-Eq-Laws.eq-mutreeF.simps sum.case-eq-if)
      using eq-class.sym by fastforce
  qed
next
  case (3 x y z)
  then show ?case
  proof (induction x arbitrary: y z)
    case (In x')
    then show ?case
      apply(cases x')
      apply (cases y; cases z; hypsubst-thin)
      apply (simp add: Derive-Eq-Laws.eq-mutreeF.simps sum.case-eq-if
eq-unit-def)
      apply (metis sum.case-eq-if)
      apply(cases y; cases z; hypsubst-thin)
      unfolding sum-set-defs prod-set-defs
      apply (simp add: Derive-Eq-Laws.eq-mutreeF.simps eq-unit-def snds.intros)
      apply (simp only: sum.case-eq-if)
      by (meson eq-class.trans)
  qed
qed

lemma eq Leaf Leaf by code-simp
lemma eq (Node (1::nat) Leaf Leaf) Leaf  $\longleftrightarrow$  False by eval
lemma eq (Node (1::nat) Leaf Leaf) (Node (1::nat) Leaf Leaf) by eval
lemma eq (Node (1::nat) (Node 2 Leaf Leaf) (Node 3 Leaf Leaf)) (Node
(1::nat) (Node 2 Leaf Leaf) (Node 4 Leaf Leaf))
 $\longleftrightarrow$  False by eval
end

```



### 3.6.2 Algebraic Classes

**theory** *Derive-Algebra-Laws*

**imports** *Main ../Derive Derive-Datatypes*  
**begin**

**datatype** *simple-int* = *A int* | *B int int* | *C*

**class** *semigroup* =  
  **fixes** *mult* :: 'a ⇒ 'a ⇒ 'a (**infixl**  $\otimes$  70)  
  **assumes** *assoc*: (x  $\otimes$  y)  $\otimes$  z = x  $\otimes$  (y  $\otimes$  z)

**class** *monoidl* = *semigroup* +  
  **fixes** *neutral* :: 'a (**1**)  
  **assumes** *neutl* : **1**  $\otimes$  x = x

**class** *group* = *monoidl* +  
  **fixes** *inverse* :: 'a ⇒ 'a  
  **assumes** *invt*: (*inverse* x)  $\otimes$  x = **1**

**definition** *semigroup-law* :: ('a ⇒ 'a ⇒ 'a) ⇒ bool **where**  
*semigroup-law* *MULT* = (∀ x y z. *MULT* (*MULT* x y) z = *MULT* x (*MULT* y z))

**definition** *monoidl-law* :: ('a ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ bool **where**  
*monoidl-law* *NEUTRAL* *MULT* = ((∀ x. *MULT* *NEUTRAL* x = x) ∧ *semigroup-law* *MULT*)

**definition** *group-law* :: ('a ⇒ 'a) ⇒ 'a ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ bool **where**  
*group-law* *INVERSE* *NEUTRAL* *MULT* = ((∀ x. *MULT* (*INVERSE* x) x = *NEUTRAL*) ∧ *monoidl-law* *NEUTRAL* *MULT*)

**lemma** *transfer-semigroup*:

**assumes** *Derive.iso* f g  
  **shows** *semigroup-law* *MULT* ⇒ *semigroup-law* (λx y. g (*MULT* (f x) (f y)))  
  **unfolding** *semigroup-law-def*  
  **using** *assms* **unfolding** *Derive.iso-def* **by** *simp*

**lemma** *transfer-monoidl*:

**assumes** *Derive.iso* f g  
  **shows** *monoidl-law* *NEUTRAL* *MULT* ⇒ *monoidl-law* (g *NEUTRAL*) (λx y. g (*MULT* (f x) (f y)))  
  **unfolding** *monoidl-law-def* *semigroup-law-def*  
  **using** *assms* **unfolding** *Derive.iso-def* **by** *simp*

**lemma** *transfer-group*:  
**assumes** *Derive.iso f g*  
**shows** *group-law INVERSE NEUTRAL MULT  $\implies$  group-law ( $\lambda x. g$   
(*INVERSE* (f x))) (g *NEUTRAL*) ( $\lambda x y. g$  (*MULT* (f x) (f y)))*  
**unfolding** *group-law-def monoidl-law-def semigroup-law-def*  
**using** *assms unfolding Derive.iso-def by simp*

**lemma** *semigroup-law-semigroup*: *semigroup-law mult*  
**unfolding** *semigroup-law-def*  
**using** *semigroup-class.axioms unfolding class.semigroup-def .*

**lemma** *monoidl-law-monoidl*: *monoidl-law neutral mult*  
**unfolding** *monoidl-law-def*  
**using** *monoidl-class.axioms semigroup-law-semigroup*  
**unfolding** *class.monoidl-axioms-def by simp*

**lemma** *group-law-group*: *group-law inverse neutral mult*  
**unfolding** *group-law-def*  
**using** *group-class.axioms monoidl-law-monoidl*  
**unfolding** *class.group-axioms-def by simp*

**derive-generic-setup** *semigroup*  
**unfolding** *semigroup-class-law-def*  
*Derive.iso-def*  
**by** *simp*

**derive-generic-setup** *monoidl*  
**unfolding** *monoidl-class-law-def semigroup-class-law-def Derive.iso-def*  
**by** *simp*

**derive-generic-setup** *group*  
**unfolding** *group-class-law-def monoidl-class-law-def semigroup-class-law-def*  
*Derive.iso-def*  
**by** *simp*

**instantiation** *int and unit:: semigroup*  
**begin**  
**definition** *mult-int-def* : *mult (x::int) y = x + y*  
**definition** *mult-unit-def*: *mult (x::unit) y = x*  
**instance proof**  
**fix** *x y z :: int*  
**show**  $x \otimes y \otimes z = x \otimes (y \otimes z)$   
**unfolding** *mult-int-def by simp*

```

next
  fix x y z :: unit
  show x ⊗ y ⊗ z = x ⊗ (y ⊗ z)
    unfolding mult-unit-def by simp
qed
end
instantiation int and unit:: monoidl
begin
  definition neutral-int-def : neutral = (0::int)
  definition neutral-unit-def: neutral = ()
instance proof
  fix x :: int
  show 1 ⊗ x = x unfolding neutral-int-def mult-int-def by simp
next
  fix x :: unit
  show 1 ⊗ x = x unfolding neutral-unit-def mult-unit-def by simp
qed
end

instantiation int and unit:: group
begin
  definition inverse-int-def : inverse (i::int) = 1 - i
  definition inverse-unit-def: inverse u = ()
instance proof
  fix x :: int
  show inverse x ⊗ x = 1 unfolding inverse-int-def mult-int-def by simp
next
  fix x :: unit
  show inverse x ⊗ x = 1 unfolding inverse-unit-def mult-unit-def by
simp
qed
end

instantiation prod and sum :: (semigroup, semigroup) semigroup
begin
  definition mult-prod-def: x ⊗ y = (fst x ⊗ fst y, snd x ⊗ snd y)
  definition mult-sum-def: x ⊗ y = (case x of Inl a ⇒ (case y of Inl b ⇒
Inl (a ⊗ b) | Inr b ⇒ Inr b)
| Inr a ⇒ (case y of Inl b ⇒ Inr a | Inr
b ⇒ Inr (a ⊗ b)))
instance proof
  fix x y z :: ('a::semigroup) × ('b::semigroup)
  show x ⊗ y ⊗ z = x ⊗ (y ⊗ z) unfolding mult-prod-def by (simp add:
assoc)

```

```

next
  fix  $x\ y\ z :: ('a::semigroup) + ('b::semigroup)$ 
  show  $x \otimes y \otimes z = x \otimes (y \otimes z)$  unfolding mult-sum-def
    by (simp add: assoc sum.case-eq-if)
qed
end

instantiation prod and sum :: (monoidl, monoidl) monoidl
begin
  definition neutral-prod-def: neutral = (neutral,neutral)
  definition neutral-sum-def: neutral = Inl neutral
instance proof
  fix  $x :: ('a::monoidl) \times ('b::monoidl)$ 
  show  $\mathbf{1} \otimes x = x$  unfolding neutral-prod-def mult-prod-def by (simp add:
neutl)
next
  fix  $x :: ('a::monoidl) + ('b::monoidl)$ 
  show  $\mathbf{1} \otimes x = x$  unfolding neutral-sum-def mult-sum-def
    by (simp add: neutl sum.case-eq-if sum.exhaust-sel)
qed
end

instantiation prod :: (group, group) group
begin
  definition inverse-prod-def: inverse p = (inverse (fst p), inverse (snd p))
instance proof
  fix  $x :: ('a::group) \times ('b::group)$ 
  show  $\text{inverse } x \otimes x = \mathbf{1}$  unfolding inverse-prod-def mult-prod-def neu-
tral-prod-def
    by (simp add: invl)
qed
end

derive-generic semigroup simple-int .
derive-generic monoidl simple-int .

derive-generic semigroup either .
derive-generic monoidl either .

lemma  $(B\ 1\ 6) \otimes (B\ 4\ 5) = B\ 4\ 11$  by eval
lemma  $(A\ 2) \otimes (A\ 3) = A\ 5$  by eval
lemma  $(B\ 1\ 6) \otimes \mathbf{1} = B\ 0\ 6$  by eval

```

**lemma**  $(L\ 3) \otimes ((L\ 4)::(int,int)\ \text{either}) = L\ 7$  **by** *eval*  
**lemma**  $(R\ (2::int)) \otimes (L\ (3::int)) = R\ 2$  **by** *eval*

**derive-generic** *semigroup list*

**proof** *goal-cases*

**case**  $(1\ x\ y\ z)$

**then show** *?case*

**proof** (*induction x arbitrary: y z*)

**case**  $(In\ x')$

**then show** *?case*

**apply** (*cases x'*)

**apply** (*cases y; cases z; hypsubst-thin*)

**apply** (*simp add: Derive-Algebra-Laws.mult-mulistF.simps sum.case-eq-if mult-unit-def*)

**apply** (*cases y; cases z; hypsubst-thin*)

**unfolding** *sum-set-defs prod-set-defs*

**apply** (*simp add: Derive-Algebra-Laws.mult-mulistF.simps mult-unit-def*)

**by** (*simp add: sum.case-eq-if assoc*)

**qed**

**qed**

**derive-generic** *semigroup tree*

**proof** *goal-cases*

**case**  $(1\ x\ y\ z)$

**then show** *?case*

**proof** (*induction x arbitrary: y z*)

**case**  $(In\ x')$

**then show** *?case*

**apply** (*cases x'*)

**apply** (*cases y; cases z; hypsubst-thin*)

**apply** (*simp add: Derive-Algebra-Laws.mult-mutreeF.simps sum.case-eq-if mult-unit-def*)

**apply** (*cases y; cases z; hypsubst-thin*)

**unfolding** *sum-set-defs prod-set-defs*

**apply** (*simp add: Derive-Algebra-Laws.mult-mutreeF.simps mult-unit-def*)

**by** (*simp add: semigroup-class.assoc sum.case-eq-if*)

**qed**

**qed**

**derive-generic** *monoidl list*

**proof** *goal-cases*

**case**  $(1\ x)$

**then show** *?case*

**proof** (*induction x*)

```

    case (In x')
    then show ?case
    apply(cases x')
    by (auto simp add: Derive-Algebra-Laws.neutral-multisetF-def sum.case-eq-if
neutral-unit-def)
qed
qed

```

**derive-generic** *monoidl tree*

**proof** *goal-cases*

```

    case (1 x)
    then show ?case
    proof (induction x)
    case (In x')
    then show ?case
    apply(cases x')
    by (auto simp add: Derive-Algebra-Laws.neutral-mutreeF-def sum.case-eq-if
neutral-unit-def)
    qed
qed

```

**lemma**  $[1,2,3,4::int] \otimes [1,2,3] = [2,4,6,4]$  **by** *eval*

**lemma**  $(Node (3::int) Leaf Leaf) \otimes (Node (1::int) Leaf Leaf) = (Node 4 Leaf Leaf)$  **by** *eval*

**end**

## References

- [1] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löb. A generic deriving mechanism for Haskell. *ACM Sigplan Notices*, 45(11):37–48, 2010.
- [2] Miles Sabin. shapeless: generic programming for Scala. <https://github.com/milessabin/shapeless>, 2018. [Online; accessed 17-April-2018].