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

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*))
⟨*proof*⟩

lemma *measure-tagged-snd*[*measure-function*]: *is-measure f* ⟹ *is-measure* (*λ p. f* (*Tagged-Prod-Sum.snd p*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *size-tagged-prod-measure*:
is-measure f ⟹ *is-measure g* ⟹ *is-measure* (*Tagged-Prod-Sum.prod.size-prod f g*)
⟨*proof*⟩

lemma *size-tagged-sum-measure*:
is-measure f ⟹ *is-measure g* ⟹ *is-measure* (*Tagged-Prod-Sum.sum.size-sum*

f g)
⟨*proof*⟩

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*: $(\bigwedge a.\ to\ (from\ a) = a) \Longrightarrow (\bigwedge b.\ from\ (to\ b) = b) \Longrightarrow$ *iso from to*
  ⟨*proof*⟩

end

⟨*ML*⟩

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 ⟨*proof*⟩
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* ⇒ *eq a b*))

instance ⟨*proof*⟩
end

derive-generic *eq simple* ⟨*proof*⟩

lemma *eq* (*A 4*) (*A 4*) ⟨*proof*⟩
lemma *eq* (*A 6*) (*A 4*) ⟷ *False* ⟨*proof*⟩
lemma *eq C C* ⟨*proof*⟩
lemma *eq* (*B 4 5*) (*B 4 5*) ⟨*proof*⟩
lemma *eq* (*B 4 4*) (*A 3*) ⟷ *False* ⟨*proof*⟩
lemma *eq C* (*A 4*) ⟷ *False* ⟨*proof*⟩

derive-generic *eq either* ⟨*proof*⟩

lemma *eq* (*L* (*3*::*nat*)) (*R 3*) ⟷ *False* ⟨*proof*⟩
lemma *eq* (*L* (*3*::*nat*)) (*L 3*) ⟨*proof*⟩
lemma *eq* (*L* (*3*::*nat*)) (*L 4*) ⟷ *False* ⟨*proof*⟩

derive-generic *eq list* ⟨*proof*⟩

lemma *eq* ([]::(*nat list*)) [] ⟨*proof*⟩
lemma *eq* ([*1*,*2*,*3*]:: (*nat list*)) [*1*,*2*,*3*] ⟨*proof*⟩
lemma *eq* [(*1*::*nat*)] [*1*,*2*] ⟷ *False* ⟨*proof*⟩

derive-generic *eq tree* ⟨*proof*⟩

lemma *eq Leaf Leaf* ⟨*proof*⟩
lemma *eq* (*Node* (*1*::*nat*) *Leaf Leaf*) *Leaf* ⟷ *False* ⟨*proof*⟩
lemma *eq* (*Node* (*1*::*nat*) *Leaf Leaf*) (*Node* (*1*::*nat*) *Leaf Leaf*) ⟨*proof*⟩
lemma *eq* (*Node* (*1*::*nat*) (*Node 2 Leaf Leaf*) (*Node 3 Leaf Leaf*)) (*Node* (*1*::*nat*) (*Node 2 Leaf Leaf*) (*Node 4 Leaf Leaf*))
⟷ *False* ⟨*proof*⟩

derive-generic *eq even-nat* ⟨*proof*⟩
derive-generic *eq exp* ⟨*proof*⟩

lemma *eq Even-Zero Even-Zero* ⟨*proof*⟩
lemma *eq Even-Zero* (*Even-Succ* (*Odd-Succ Even-Zero*)) ⟷ *False* ⟨*proof*⟩

lemma *eq* (*Odd-Succ* (*Even-Succ* (*Odd-Succ Even-Zero*))) (*Odd-Succ* (*Even-Succ* (*Odd-Succ Even-Zero*))) ⟨*proof*⟩
lemma *eq* (*Odd-Succ* (*Even-Succ* (*Odd-Succ Even-Zero*))) (*Odd-Succ* (*Even-Succ* (*Odd-Succ* (*Even-Succ* (*Odd-Succ Even-Zero*)))))
  ⟷ *False* ⟨*proof*⟩

lemma *eq* (*Const* (*1*::*nat*)) (*Const* (*1*::*nat*)) ⟨*proof*⟩
lemma *eq* (*Const* (*1*::*nat*)) (*Var* (*1*::*nat*)) ⟷ *False* ⟨*proof*⟩
lemma *eq* (*Term* (*Prod* (*Const* (*1*::*nat*)) (*Factor* (*Const* (*2*::*nat*))))) (*Term* (*Prod* (*Const* (*1*::*nat*)) (*Factor* (*Const* (*2*::*nat*)))))
  ⟨*proof*⟩
lemma *eq* (*Term* (*Prod* (*Const* (*1*::*nat*)) (*Factor* (*Const* (*2*::*nat*))))) (*Term* (*Prod* (*Const* (*1*::*nat*)) (*Factor* (*Const* (*3*::*nat*)))))
  ⟷ *False* ⟨*proof*⟩

end

3.3 Encoding

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

class *encodeable* =
 fixes *encode* :: *'a* ⇒ *bool list*

instantiation *nat* **and** *unit*:: *encodeable*
begin
 fun *encode-nat* :: *nat* ⇒ *bool list* **where**
 encode-nat 0 = [] |
 encode-nat (*Suc n*) = *True* # (*encode n*)

 definition *encode-unit*: *encode* (*x*::*unit*) = []
 instance ⟨*proof*⟩
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* ⇒ *False* # *encode a*
  | *Inr a* ⇒ *True* # *encode a*)

instance ⟨*proof*⟩
end

derive-generic *encodeable simple* ⟨*proof*⟩
derive-generic *encodeable either* ⟨*proof*⟩

lemma *encode* (*B 3 4*) = [*True*, *False*, *True*, *True*, *True*, *True*, *True*, *True*, *True*] ⟨*proof*⟩
lemma *encode C* = [*True*, *True*] ⟨*proof*⟩
lemma *encode* (*R* (*3*::*nat*)) = [*True*, *True*, *True*, *True*] ⟨*proof*⟩

derive-generic *encodeable list* ⟨*proof*⟩
derive-generic *encodeable tree* ⟨*proof*⟩

lemma *encode* [*1*,*2*,*3*,*4*::*nat*]
= [*True*, *True*, *True*, *True*, *True*, *True*, *True*, *True*, *True*, *True*, *True*, *True*, *True*, *True*, *False*] ⟨*proof*⟩
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*] ⟨*proof*⟩

derive-generic *encodeable even-nat* ⟨*proof*⟩
derive-generic *encodeable exp* ⟨*proof*⟩

lemma *encode* (*Odd-Succ* (*Even-Succ* (*Odd-Succ Even-Zero*)))
= [*True*, *False*, *True*, *True*, *False*, *False*] ⟨*proof*⟩
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*]
⟨*proof*⟩

end

3.4 Algebraic Classes

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

class *semigroup* =

fixes *mult* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\otimes$ *70*)

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

class *group* = *monoidl* +
fixes *inverse* :: $'a \Rightarrow 'a$

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 ⟨*proof*⟩
end
instantiation *nat* **and** *unit*:: *monoidl*
begin
definition *neutral-nat* : *neutral* = (*0*::*nat*)
definition *neutral-unit-def*: *neutral* = ()
instance ⟨*proof*⟩
end

instantiation *nat* **and** *unit*:: *group*
begin
definition *inverse-nat* : *inverse* (*i*::*nat*) = **1** − *i*
definition *inverse-unit-def*: *inverse u* = ()
instance ⟨*proof*⟩
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* $\Rightarrow$ (*case y of Inl b* $\Rightarrow$ *Inl* ($a \otimes b$) | *Inr b* $\Rightarrow$ *Inl a*)
| *Inr a* $\Rightarrow$ (*case y of Inl b* $\Rightarrow$ *Inr a* | *Inr b* $\Rightarrow$ *Inr* ($a \otimes b$)))
instance ⟨*proof*⟩
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*⟩
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* ⇒ *Inr* (*inverse b*))
instance ⟨*proof*⟩
end

derive-generic *semigroup simple* ⟨*proof*⟩
derive-generic *monoidl simple* ⟨*proof*⟩
derive-generic *group simple* ⟨*proof*⟩

lemma (*B* **1** *6*) ⊗ (*B 4 5*) = *B 4 11* ⟨*proof*⟩
lemma (*A 2*) ⊗ (*A 3*) = *A 5* ⟨*proof*⟩
lemma (*B* **1** *6*) ⊗ **1** = *B 0 6* ⟨*proof*⟩

derive-generic *group either* ⟨*proof*⟩

lemma (*L 3*) ⊗ ((*L 4*)::(*nat*,*nat*) *either*) = *L 7* ⟨*proof*⟩
lemma (*R* (*2*::*nat*)) ⊗ (*L* (*3*::*nat*)) = *R 2* ⟨*proof*⟩

derive-generic *semigroup list* ⟨*proof*⟩
derive-generic *monoidl list* ⟨*proof*⟩
derive-generic *group list* ⟨*proof*⟩
derive-generic *semigroup tree* ⟨*proof*⟩
derive-generic *monoidl tree* ⟨*proof*⟩
derive-generic *group tree* ⟨*proof*⟩

lemma [*1*,*2*,*3*,*4*::*nat*] ⊗ [*1*,*2*,*3*] = [*2*,*4*,*6*,*4*] ⟨*proof*⟩
lemma *inverse* [*1*,*2*,*3*::*nat*] = [*0*,*0*,*0*] ⟨*proof*⟩

derive-generic *semigroup even-nat* ⟨*proof*⟩
derive-generic *monoidl even-nat* ⟨*proof*⟩
derive-generic *group even-nat* ⟨*proof*⟩
derive-generic *semigroup exp* ⟨*proof*⟩

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 ⟨*proof*⟩
end

⟨*ML*⟩

derive-generic *group exp* ⟨*proof*⟩

lemma (*Odd-Succ* (*Even-Succ* (*Odd-Succ Even-Zero*))) ⊗ (*Odd-Succ Even-Zero*)

    = *Odd-Succ* (*Even-Succ* (*Odd-Succ Even-Zero*)) ⟨*proof*⟩
lemma *inverse* (*Odd-Succ Even-Zero*) = *Odd-Succ Even-Zero* ⟨*proof*⟩
lemma (*Term* (*Prod* ((*Const 1*)::(*nat*, *nat*) *fct*) (*Factor* (*Const* (*2*::*nat*)))))

  ⊗ (*Term* (*Prod* (*Const* (*2*::*nat*)) (*Factor* ((*Const 2*)::(*nat*, *nat*) *fct*))))
  = *Term* (*Prod* (*Const 3*) (*Factor* (*Const 4*))) ⟨*proof*⟩

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 ⟨*proof*⟩
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 ⟨*proof*⟩
end

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

derive-generic (*metadata*) *showable simple* ⟨*proof*⟩
derive-generic (*metadata*) *showable either* ⟨*proof*⟩

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

derive-generic (*metadata*) *showable list* ⟨*proof*⟩
derive-generic (*metadata*) *showable tree* ⟨*proof*⟩

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

derive-generic (*metadata*) *showable even-nat* ⟨*proof*⟩
derive-generic (*metadata*) *showable exp* ⟨*proof*⟩

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* :: $'a \Rightarrow 'a \Rightarrow bool$
  assumes *refl*: *eq x x* **and**
    sym: $eq\ x\ y \Longrightarrow eq\ y\ x$ **and**
    trans: $eq\ x\ y \Longrightarrow eq\ y\ z \Longrightarrow eq\ x\ z$

derive-generic-setup *eq*
  ⟨*proof*⟩

lemma *eq-law-eq*: *eq-class-law eq*
  ⟨*proof*⟩

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* ⟷ *True*
instance ⟨*proof*⟩
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* ⇒ *eq a b*))
instance ⟨*proof*⟩
end

derive-generic *eq simple* ⟨*proof*⟩

lemma *eq* (*A 4*) (*A 4*) ⟨*proof*⟩
lemma *eq* (*A 6*) (*A 4*) ⟷ *False* ⟨*proof*⟩
lemma *eq C C* ⟨*proof*⟩
lemma *eq* (*B 4 5*) (*B 4 5*) ⟨*proof*⟩
lemma *eq* (*B 4 4*) (*A 3*) ⟷ *False* ⟨*proof*⟩
lemma *eq C* (*A 4*) ⟷ *False* ⟨*proof*⟩

derive-generic *eq either* ⟨*proof*⟩

lemma *eq* (*L* (*3*::*nat*)) (*R 3*) ⟷ *False* ⟨*proof*⟩
lemma *eq* (*L* (*3*::*nat*)) (*L 3*) ⟨*proof*⟩
lemma *eq* (*L* (*3*::*nat*)) (*L 4*) ⟷ *False* ⟨*proof*⟩

derive-generic *eq list*
⟨*proof*⟩

lemma *eq* ([]::(*nat list*)) [] ⟨*proof*⟩
lemma *eq* ([*1*,*2*,*3*]:: (*nat list*)) [*1*,*2*,*3*] ⟨*proof*⟩
lemma *eq* [(*1*::*nat*)] [*1*,*2*] ⟷ *False* ⟨*proof*⟩

derive-generic *eq tree*
⟨*proof*⟩

lemma *eq Leaf Leaf* ⟨*proof*⟩
lemma *eq* (*Node* (*1*::*nat*) *Leaf Leaf*) *Leaf* ⟷ *False* ⟨*proof*⟩
lemma *eq* (*Node* (*1*::*nat*) *Leaf Leaf*) (*Node* (*1*::*nat*) *Leaf Leaf*) ⟨*proof*⟩
lemma *eq* (*Node* (*1*::*nat*) (*Node 2 Leaf Leaf*) (*Node 3 Leaf Leaf*)) (*Node* (*1*::*nat*) (*Node 2 Leaf Leaf*) (*Node 4 Leaf Leaf*))
⟷ *False* ⟨*proof*⟩
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** ⊗ *70*)
assumes *assoc*: (*x* ⊗ *y*) ⊗ *z* = *x* ⊗ (*y* ⊗ *z*)

class *monoidl* = *semigroup* +
fixes *neutral* :: ′*a* (**1**)
assumes *neutl* : **1** ⊗ *x* = *x*

class *group* = *monoidl* +
fixes *inverse* :: ′*a* ⇒ ′*a*
assumes *invl*: (*inverse x*) ⊗ *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* (λ*x y*. *g* (*MULT* (*f x*) (*f y*)))
⟨*proof*⟩

lemma *transfer-monoidl*:
assumes *Derive.iso f g*
shows *monoidl-law NEUTRAL MULT* $\Longrightarrow$ *monoidl-law* (*g NEUTRAL*) ($\lambda x\ y.\ g$ (*MULT* (*f x*) (*f y*)))
⟨*proof*⟩

lemma *transfer-group*:
assumes *Derive.iso f g*
shows *group-law INVERSE NEUTRAL MULT* $\Longrightarrow$ *group-law* (λ *x*. *g* (*INVERSE* (*f x*))) (*g NEUTRAL*) ($\lambda x\ y.\ g$ (*MULT* (*f x*) (*f y*)))
⟨*proof*⟩

lemma *semigroup-law-semigroup*: *semigroup-law mult*
⟨*proof*⟩

lemma *monoidl-law-monoidl*: *monoidl-law neutral mult*
⟨*proof*⟩

lemma *group-law-group*: *group-law inverse neutral mult*
⟨*proof*⟩

derive-generic-setup *semigroup*
⟨*proof*⟩

derive-generic-setup *monoidl*
⟨*proof*⟩

derive-generic-setup *group*
⟨*proof*⟩

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*⟩
end
instantiation *int* **and** *unit*:: *monoidl*
begin
definition *neutral-int-def* : *neutral* = (*0*::*int*)
definition *neutral-unit-def*: *neutral* = ()
instance ⟨*proof*⟩
end

instantiation *int* **and** *unit*:: *group*
begin
  definition *inverse-int-def* : *inverse* (*i*::*int*) = **1** − *i*
  definition *inverse-unit-def*: *inverse u* = ()
instance ⟨*proof*⟩
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*⟩
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*⟩
end

instantiation *prod* :: (*group*, *group*) *group*
begin
  definition *inverse-prod-def*: *inverse p* = (*inverse* (*fst p*), *inverse* (*snd p*))
instance ⟨*proof*⟩
end

derive-generic *semigroup simple-int* ⟨*proof*⟩
derive-generic *monoidl simple-int* ⟨*proof*⟩

derive-generic *semigroup either* ⟨*proof*⟩
derive-generic *monoidl either* ⟨*proof*⟩

lemma (*B* **1** *6*) ⊗ (*B 4 5*) = *B 4 11* ⟨*proof*⟩
lemma (*A 2*) ⊗ (*A 3*) = *A 5* ⟨*proof*⟩
lemma (*B* **1** *6*) ⊗ **1** = *B 0 6* ⟨*proof*⟩

lemma (*L 3*) ⊗ ((*L 4*)::(*int*,*int*) *either*) = *L 7* ⟨*proof*⟩
lemma (*R* (*2*::*int*)) ⊗ (*L* (*3*::*int*)) = *R 2* ⟨*proof*⟩

derive-generic *semigroup list*
⟨*proof*⟩

derive-generic *semigroup tree*
⟨*proof*⟩

derive-generic *monoidl list*
⟨*proof*⟩

derive-generic *monoidl tree*
⟨*proof*⟩

lemma $[1,2,3,4::int] \otimes [1,2,3] = [2,4,6,4]$ ⟨*proof*⟩
lemma $(Node\ (3::int)\ Leaf\ Leaf) \otimes (Node\ (1::int)\ Leaf\ Leaf) = (Node\ 4\ Leaf\ Leaf)$ ⟨*proof*⟩

end

References

[1] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. 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].