Haskell’s Show-Class in Isabelle/HOL*

Christian Sternagel  René Thiemann

May 28, 2015

Abstract

We implemented a type-class for pretty-printing, similar to Haskell’s Show-class [1]. Moreover, we provide instantiations for Isabelle/HOL’s standard types like $\mathbb{B}$, prod, sum, N, Z, and Q. It is further possible to automatically derive “to-string” functions for arbitrary user defined datatypes similar to Haskell’s “deriving Show”.

Contents

1 Converting Arbitrary Values to Readable Strings 1

1.1 The Show-Law 2

1.2 Show-Functions for Characters and Strings 4

2 Instances of the Show Class for Standard Types 6

1 Converting Arbitrary Values to Readable Strings

A type class similar to Haskell’s Show class, allowing for constant-time concatenation of strings using function composition.

theory Show
imports
  Main
  ../Deriving/Generator-Aux
  ../Deriving/Derive-Manager
begin

  type-synonym
    shows = string ⇒ string

  — show-functions with precedence
  typedecl
    'a showsp = nat ⇒ 'a ⇒ shows

*This research is supported by FWF (Austrian Science Fund) projects J3202 and P22767.
1.1 The Show-Law

The "show law", \( \texttt{shows-prec} \ p \ x \ (r \ @ \ s) = \texttt{shows-prec} \ p \ x \ r \ @ \ s \), states that show-functions do not temper with or depend on output produced so far.

**named-theorems** show-law-simps (simplification rules for proving the show law)

**named-theorems** show-law-intros (introduction rules for proving the show law)

**definition** show-law :: 'a showsp \Rightarrow \ 'a \Rightarrow \ bool

where
show-law \ s \ x \ \leftrightarrow \ (\forall \ p \ y \ z. \ s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z)

**lemma** show-lawI:
show-law \ s \ x \ \Rightarrow \ (\forall \ p \ y \ z. \ s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z) \ \Rightarrow \ \text{show-law} \ s \ x

**lemma** show-lawE:
show-law \ s \ x \ \Rightarrow \ (s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z) \ \Rightarrow \ P

**lemma** show-lawD:
show-law \ s \ x \ \Rightarrow \ s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z

**class** show =

*fixes* shows-prec :: 'a showsp  
and shows-list :: 'a list \Rightarrow \ shows

*assumes* shows-prec-append [show-law-simps]: shows-prec \ p \ x \ (r \ @ \ s) = shows-prec \ p \ x \ r \ @ \ s  
and
shows-list-append [show-law-simps]: shows-list \ xs \ (r \ @ \ s) = shows-list \ xs \ r \ @ \ s

**begin**

**abbreviation** shows x \equiv \texttt{shows-prec} \ 0 \ x

**abbreviation** show x \equiv \texttt{shows} \ x \ ""

**end**

Convert a string to a show-function that simply prepends the string unchanged.

**definition** shows-string :: string \Rightarrow \ shows

where
shows-string = \op \ @

**lemma** shows-string-append [show-law-simps]:
shows-string \ x \ (r \ @ \ s) = shows-string \ x \ r \ @ \ s

**fun** shows-sep :: ('a \Rightarrow \ shows) \Rightarrow \ shows \Rightarrow \ 'a list \Rightarrow \ shows

where
shows-sep \ s \ sep \ [] = shows-string \ "" |
shows-sep s sep [x] = s x |
shows-sep s sep (x#xs) = s x o sep o shows-sep s sep xs

**lemma** shows-sep-append [show-law-simps]:

assumes \( \bigwedge r \ s. \forall x \in \text{set} \ xs. \text{showsx} x (r \@ s) = \text{showsx} x r \@ s \)

and \( \bigwedge r. \ s. \text{sep} (r \@ s) = \text{sep} r \@ s \)

shows shows-sep showsx sep xs (r \@ s) = shows-sep showsx sep xs r \@ s

⟨proof⟩

**lemma** shows-sep-map:

shows-sep f sep (map g xs) = shows-sep (f o g) sep xs

⟨proof⟩

**definition**

shows-list-gen :: (′a ⇒ shows) ⇒ string ⇒ string ⇒ string ⇒ string ⇒ ′a list ⇒ shows

where

shows-list-gen showsx e l s r xs =
(if xs = [] then shows-string e
  else shows-string l o shows-sep showsx (shows-string s) xs o shows-string r)

**lemma** shows-list-gen-append [show-law-simps]:

assumes \( \bigwedge r. \forall x \in \text{set} \ xs. \text{showsx} x (r \@ s) = \text{showsx} x r \@ s \)

shows shows-list-gen showsx e l sep r xs (s \@ t) = shows-list-gen showsx e l sep r xs s \@ t

⟨proof⟩

**lemma** shows-list-gen-map:

shows-list-gen f e l sep r (map g xs) = shows-list-gen (f o g) e l sep r xs

⟨proof⟩

**definition** pshowsp-list :: nat ⇒ shows list ⇒ shows

where

pshowsp-list p xs = shows-list-gen id (′[]) (′[]′) (′[]′) (′[]′) xs

**definition** showsp-list :: ′a showsp ⇒ nat ⇒ ′a list ⇒ shows

where

[code del]: showsp-list s p = pshowsp-list p o map (s 0)

**lemma** showsp-list-code [code]:

shows-list-gen s p xs = shows-list-gen (s 0) (′[]′) (′[]′) (′[]′) (′[]′) xs

⟨proof⟩

**lemma** show-law-list [show-law-intros]:

(\( \bigwedge x. x \in \text{set} \ xs \implies \text{show-law} s x \) \implies \text{show-law} (shows-list-gen s) xs)

⟨proof⟩

**lemma** showsp-list-append [show-law-simps]:

(\( \bigwedge p \ y \ z. \forall x \in \text{set} \ xs. s \ p \ x (y \@ z) = s \ p \ x \ y \@ z \) \implies
shows-list s p xs (y @ z) = shows-list s p xs y @ z
(proof)

1.2 Show-Functions for Characters and Strings

instantiation char :: show
begin
definition shows-prec p (c::char) = op # c
definition shows-list (cs::string) = shows-string cs
instance ⟨proof⟩
end
definition shows-nl = shows (CHR ‘←’)
definition shows-space = shows (CHR ‘’)
definition shows-paren s = shows (CHR ‘(‘) o s o shows (CHR ‘)’)
definition shows-quote s = shows (Char Nibble2 Nibble7) o s o shows (Char Nibble2 Nibble7)
abbreviation apply-if b s ≡ (if b then s else id) — conditional function application
Parenthesize only if precedence is greater than 0.
definition shows-pl (p::nat) = apply-if (p > 0) (shows (CHR ‘(‘))
definition shows-pr (p::nat) = apply-if (p > 0) (shows (CHR ‘)’))
lemma shows-nl-append [show-law-simps]: shows-nl (x @ y) = shows-nl x @ y and
shows-space-append [show-law-simps]: shows-space (x @ y) = shows-space x @ y and
shows-paren-append [show-law-simps]:
(∀ x y. s (x @ y) = s x @ y) ⇒ shows-paren s (x @ y) = shows-paren s x @ y and
shows-quote-append [show-law-simps]:
(∀ x y. s (x @ y) = s x @ y) ⇒ shows-quote s (x @ y) = shows-quote s x @ y and
shows-pl-append [show-law-simps]: shows-pl p (x @ y) = shows-pl p x @ y and
shows-pr-append [show-law-simps]: shows-pr p (x @ y) = shows-pr p x @ y
(proof)
lemma o-append:
(∀ x y. f (x @ y) = f x @ y) ⇒ g (x @ y) = g x @ y ⇒ (f o g) (x @ y) = (f o g) x @ y
(proof)
⟨ML⟩

instantiation list :: (show) show
begin
definition shows-prec (p :: nat) (xs :: 'a list) = shows-list xs

4
definition shows-list :: 'a list list) = showsp-list shows-pc 0 xs
instance (proof)
end

definition shows-lines :: 'a::show list ⇒ shows
where
  shows-lines = shows-sep shows shows-nl

definition shows-many :: 'a::show list ⇒ shows
where
  shows-many = shows-sep shows id

definition shows-words :: 'a::show list ⇒ shows
where
  shows-words = shows-sep shows shows-space

lemma shows-lines-append [show-law-simps]:
  shows-lines xs (r @ s) = shows-lines xs r @ s
  ⟨proof⟩

lemma shows-many-append [show-law-simps]:
  shows-many zs (r @ s) = shows-many zs r @ s
  ⟨proof⟩

lemma shows-words-append [show-law-simps]:
  shows-words xs (r @ s) = shows-words xs r @ s
  ⟨proof⟩

lemma shows-foldr-append [show-law-simps]:
  assumes \( \forall r s. \forall x \in \text{set } xs. \text{show } x (r @ s) = \text{show } x r @ s \)
  shows foldr showx xs (r @ s) = foldr showx xs r @ s
  ⟨proof⟩

lemma shows-sep-cong [fundef-cong]:
  assumes xs = ys and \( \forall x. x \in \text{set } ys \implies f x = g x \)
  shows shows-sep f sep zs = shows-sep g sep ys
  ⟨proof⟩

lemma shows-list-gen-cong [fundef-cong]:
  assumes xs = ys and \( \forall x. x \in \text{set } ys \implies f x = g x \)
  shows shows-list-gen f e l sep r xs = shows-list-gen g e l sep r ys
  ⟨proof⟩

lemma showsp-list-cong [fundef-cong]:
  xs = ys \implies p = q \implies
  (\( \forall x. x \in \text{set } ys \implies f p x = g p x \) \implies showsp-list f p xs = showsp-list g q ys
  ⟨proof⟩

5
abbreviation (input) shows-cons :: string ⇒ shows ⇒ shows (infixr +# 10)
where
  s +# p ≡ shows-string s ∘ p

abbreviation (input) shows-append :: shows ⇒ shows ⇒ shows (infixr +@+ 10)
where
  s +@+ p ≡ s ∘ p

  Don’t use Haskell’s existing ”Show” class for code-generation, since it is
  not compatible to the formalized class.

code-reserved Haskell Show

end

2 Instances of the Show Class for Standard Types

theory Show-Instances
imports
  Show ~~/src/HOL/Rat
begin

definition showsp-unit :: unit showsp
where
  showsp-unit p x = shows-string "()"

lemma show-law-unit [show-law-intros]:
  show-law showsp-unit x
  ⟨proof⟩

primrec showsp-bool :: bool showsp
where
  showsp-bool p True = shows-string "True" |
  showsp-bool p False = shows-string "False"

lemma show-law-bool [show-law-intros]:
  show-law showsp-bool x
  ⟨proof⟩

primrec pshowsp-prod :: (shows × shows) showsp
where
  pshowsp-prod p (x, y) = shows-string "(" o x o shows-string "," o y o shows-string ")"

definition showsp-prod :: 'a showsp ⇒ 'b showsp ⇒ ('a × 'b) showsp
where
  [code del]: showsp-prod s1 s2 p = pshowsp-prod p o map-prod (s1 0) (s2 0)
lemma showsp-prod-simps [simp, code]:
showsp-prod s1 s2 p (x, y) = 
  shows-string "(" o s1 0 x o shows-string "," o s2 0 y o shows-string ")"
⟨proof⟩
lemma show-law-prod [show-law-intros]:
  (∀x. x ∈ Basic-BNFs.fsts y ⇒ show-law s1 x) ⇒
  (∀x. x ∈ Basic-BNFs.snds y ⇒ show-law s2 x) ⇒
  show-law (showsp-prod s1 s2) y
⟨proof⟩

fun string-of-digit :: nat ⇒ string
where
  string-of-digit n =
  (if n = 0 then "0"
   else if n = 1 then "1"
   else if n = 2 then "2"
   else if n = 3 then "3"
   else if n = 4 then "4"
   else if n = 5 then "5"
   else if n = 6 then "6"
   else if n = 7 then "7"
   else if n = 8 then "8"
   else "9")

fun showsp-nat :: nat showsp
where
  showsp-nat p n =
  (if n < 10 then shows-string (string-of-digit n)
   else showsp-nat p (n div 10) o shows-string (string-of-digit (n mod 10)))
declare showsp-nat.simps [simp del]

lemma show-law-nat [show-law-intros]:
  show-law showsp-nat n
⟨proof⟩
lemma showsp-nat-append [show-law-simps]:
  showsp-nat p n (x @ y) = showsp-nat p n x @ y
⟨proof⟩

definition showsp-int :: int showsp
where
  showsp-int p i =
  (if i < 0 then shows-string "−" o showsp-nat p (nat (− i)) else showsp-nat p (nat i))
lemma show-law-int [show-law-intros]:
  show-law showsp-int i
⟨proof⟩
lemma showsp-int-append [show-law-simps]:
showsp-int p i (x @ y) = showsp-int p i x @ y
(proof)

definition showsp-rat :: rat showsp
where
  showsp-rat p x =
    (case quotient-of x of (d, n) ⇒
      if n = 1 then showsp-int p d else showsp-int p d o shows-string "/" o showsp-int p n)

lemma show-law-rat [show-law-intros]:
  show-law showsp-rat r
(proof)

lemma showsp-rat-append [show-law-simps]:
  showsp-rat p r (x @ y) = showsp-rat p r x @ y
(proof)

Automatic show functions are not used for unit, prod, and numbers: for unit and prod, we do not want to display "Unity" and "Pair"; for nat, we do not want to display "Suc (Suc (... (Suc 0) ...))"; and neither int nor rat are datatypes.

⟨ML⟩
derive show option sum prod unit bool nat int rat

export-code
  shows-prec :: 'a::show option showsp
  shows-prec :: ('a::show, 'b::show) sum showsp
  shows-prec :: ('a::show × 'b::show) showsp
  shows-prec :: unit showsp
  shows-prec :: bool showsp
  shows-prec :: nat showsp
  shows-prec :: int showsp
  shows-prec :: rat showsp

checking

end

References