Haskell’s Show-Class in Isabelle/HOL*

Christian Sternagel René Thiemann

September 19, 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, $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$. 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

type-synonym
   '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", \( \text{shows-prec} \ p \ x \ (r \ @ \ s) = \text{shows-prec} \ p \ x \ r \ @ \ s \), 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
\[\text{show-law} \ s \ x \leftarrow (\forall \ p \ y \ z. \ s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z)\]

**lemma** show-lawI:
\[ (\forall p y z. s p x (y @ z) = s p x y @ z) \Longrightarrow \text{show-law} \ s \ x \]
\[\langle \text{proof}\rangle\]

**lemma** show-lawE:
\[ \text{show-law} \ s \ x \Longrightarrow (s p x (y @ z) = s p x y @ z \Longrightarrow P) \Longrightarrow P \]
\[\langle \text{proof} \rangle\]

**lemma** show-lawD:
\[ \text{show-law} \ s \ x \Longrightarrow s p x (y @ z) = s p x y @ z \]
\[\langle \text{proof}\rangle\]

**class** show =
  **fixes** shows-prec :: 'a showsp
  and shows-list :: 'a list \Rightarrow shows
  **assumes** shows-prec-append [show-law-simps]: \text{shows-prec} \ p \ x \ (r \ @ \ s) = \text{shows-prec} \ p \ x \ r \ @ \ s
  and
  shows-list-append [show-law-simps]: \text{shows-list} \ xs \ (r \ @ \ s) = \text{shows-list} \ xs \ r \ @ \ s
begin

**abbreviation** shows x ≡ \text{shows-prec} \ 0 \ x
**abbreviation** show x ≡ shows x """

end

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

**definition** shows-string :: string \Rightarrow shows
where
\[ \text{shows-string} = \text{op} \ @ \]

**lemma** shows-string-append [show-law-simps]:
\[ \text{shows-string} \ x \ (r \ @ \ s) = \text{shows-string} \ x \ r \ @ \ s \]
\[\langle \text{proof}\rangle\]

**fun** shows-sep :: ('a \Rightarrow shows) \Rightarrow shows \Rightarrow 'a list \Rightarrow shows
where
\[ \text{shows-sep} \ s \ sep \ [] = \text{shows-string} """

\]

2
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 ∀ r s. ∀ x ∈ set xs. showsx x (r @ s) = showsx x r @ s
and ∀ r s. sep (r @ s) = 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 sep 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 ∀ r s. ∀ x ∈ set xs. showsx x (r @ s) = 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 shows-list :: 'a shows ⇒ nat ⇒ 'a list ⇒ shows
where
[code del]: shows-list s p = pshowsp-list p o map (s 0)

lemma shows-list-code [code]:
shows-list s p xs = shows-list-gen (s 0) "['']" "['']" "['']" "['']" xs
⟨proof⟩

lemma show-law-list [show-law-intros]:
(∀ x. x ∈ set xs ⇒ show-law s x) ⇒ show-law (shows-list s) xs
⟨proof⟩

lemma shows-list-append [show-law-simps]:
(∀ p y z. ∀ x ∈ set xs. s p x (y @ z) = s p x y @ z) ⇒
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]:

(\(\forall x y. s (x @ y) = s x @ y\) \implies shows-paren s (x @ y) = shows-paren s x @ y)

and

shows-quote-append [show-law-simps]:

(\(\forall x y. s (x @ y) = s x @ y\) \implies 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:

(\(\forall x y. f (x @ y) = f x @ y\) \implies g (x @ y) = g x @ y \implies (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
definition shows-list (xss :: 'a list list) = showsp-list shows-prec 0 xss

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 xs (r @ s) = shows-many xs 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 ∀ r s. ∀ x ∈ set xs. showx x (r @ s) = showx 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 ∀ x. x ∈ set ys ⇒ f x = g x
  shows shows-sep f sep xs = shows-sep g sep ys
  ⟨proof⟩

lemma shows-list-gen-cong [fundef-cong]:
  assumes xs = ys and ∀ x. x ∈ set ys ⇒ 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 ⇒ p = q ⇒
\( (\forall p \cdot x \in \text{set } ys \implies f \, p \, x = g \, p \, x) \implies \text{showsp-list } f \, p \, xs = \text{showsp-list } g \, q \, ys \) (proof)

**abbreviation** (input) shows-cons :: string \(\Rightarrow\) shows \(\Rightarrow\) shows (infixr \(\#+\) 10)
**where**
\( s \, +\#\ + \, p \equiv \text{shows-string } s \circ p \)

**abbreviation** (input) shows-append :: shows \(\Rightarrow\) shows \(\Rightarrow\) shows (infixr \(\oplus\) 10)
**where**
\( s \, +\oplus\ + \, p \equiv s \circ p \)

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

**code-reserved** Haskell Show

# 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 \( p \, x \equiv \text{shows-string } "()" \)

lemma show-law-unit [show-law-intros]:
  show-law showsp-unit \( x \)
(proof)

abbreviation showsp-char :: char showsp
**where**
showsp-char \( \equiv \text{shows-prec} \)

lemma show-law-char [show-law-intros]:
  show-law showsp-char \( x \)
(proof)

primrec showsp-bool :: bool showsp
**where**
showsp-bool \( p \, \text{True} = \text{shows-string } "True" \)
showsp-bool \( p \, \text{False} = \text{shows-string } "False" \)

lemma show-law-bool [show-law-intros]:
  show-law showsp-bool \( x \)
(proof)
primrec pshowsp-prod :: (shows × shows) → shows
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 → shows
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 :: char showsp  
shows-prec :: bool showsp  
shows-prec :: nat showsp  
shows-prec :: int showsp  
shows-prec :: rat showsp

**checking**
References