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 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

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 ⇒ 'a ⇒ bool

where

\[
\text{show-law } s x \leftrightarrow (\forall p y z. s p x (y @ z) = s p x y @ z)
\]

**lemma** show-lawI:

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

by (simp add: show-law-def)

**lemma** show-lawE:

show-law s x =⇒ (s p x (y @ z) = s p x y @ z =⇒ P) =⇒ P

by (auto simp: show-law-def)

**lemma** show-lawD:

show-law s x =⇒ s p x (y @ z) = s p x y @ z

by (blast elim: show-lawE)

**class** show =

fixes shows-prec :: 'a showsp

and shows-list :: 'a list ⇒ 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 ≡ 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 ⇒ shows

where

shows-string = op @

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

shows-string x (r @ s) = shows-string x r @ s

by (simp add: shows-string-def)

**fun** shows-sep :: ('a ⇒ shows) ⇒ shows ⇒ 'a list ⇒ 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** \( \forall r s. \forall x \in \text{set} \ xs. \ showsx x (r \ @ \ s) = showsx x r \ @ \ s \)

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

**using** assms

**proof** (induct xs)

**case** (Cons x xs) **then show** ?case **by** (cases xs) (simp-all)

**qed** (simp add: show-law-simps)

**lemma** shows-sep-map:

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

**by** (induct xs) (simp, case-tac xs, simp-all)

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

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

**using** assms **by** (cases xs) (simp-all add: shows-list-gen-def show-law-simps)

**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

**by** (simp-all add: shows-list-gen-def shows-sep-map)

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

**where**

pshows-list p xs = shows-list-gen id ""]" """" "" "" "" xs

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

**where**

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

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

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

**by** (simp add: showsp-list-def showsp-list-def shows-list-gen-map)

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

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

**by** (simp add: show-law-def showsp-list-code show-law-simps)
lemma showsp-list-append [show-law-simps]:
(∀ p y z. ∀ x ∈ set xs. s p x (y @ z) = s p x y @ z) ⟹
showsp-list s p xs (y @ z) = showsp-list s p xs y @ z
by (simp add: show-law-simps showsp-list-def pshowsp-list-def)

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 by (default) (simp-all add: shows-prec-char-def shows-list-char-def show-law-simps)
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
by (simp-all add: shows-nl-def shows-space-def shows-paren-def shows-quote-def
shows-pl-def shows-pr-def show-law-simps)

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
by simp

ML-file show-generator.ML
local-setup ⟨⟨
Show-Generator.register-foreign-partial-and-full-showsp @\{type-name list\} 0
@\{term pshowsp-list\}
@\{term showsp-list\} (SOME @\{thm showsp-list-def\})
@\{term map\} (SOME @\{thm list.map-comp\}) [true] @\{thm show-law-list\} ⟩⟩

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 by (default) (simp-all add: show-law-simps shows-prec-list-def shows-list-list-def)

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
by (simp add: shows-lines-def show-law-simps)

lemma shows-many-append [show-law-simps]:
shows-many xs (r @ s) = shows-many xs r @ s
by (simp add: shows-many-def show-law-simps)

lemma shows-words-append [show-law-simps]:
shows-words xs (r @ s) = shows-words xs r @ s
by (simp add: shows-words-def show-law-simps)

lemma shows-foldr-append [show-law-simps]:
assumes f r s. ∀x ∈ set xs. showx x (r @ s) = showx x r @ s
shows foldr showx xs (r @ s) = foldr showx xs r @ s
using assms by (induct xs) (simp-all)

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
using assms

proof (induct ys arbitrary: xs)
  case (Cons y ys)
  then show ?case by (cases ys) simp
qed simp

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
using shows-sep-cong [of xs ys f g] assms by (cases xs) (auto simp: shows-list-gen-def)

lemma shows-sp-list-cong [fundef-cong]:
  \( \forall p. x \in \text{set} \ ys \implies f p x = g p x \) \( \implies \) shows-sp-list f p xs = shows-sp-list g q ys
by (simp add: shows-sp-list-code cong: shows-list-gen-cong)

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 "\@" 10)
where
  \( s + \@ + p \equiv s \circ p \)

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

code-reserved Haskell Show

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
by (rule show-lawI) (simp add: showsp-unit-def show-law-simps)

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
  by (rule show-lawI, cases x) (simp-all add: show-law-simps)

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 ""
  by (simp add: showsp-prod-def)

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 (induct y)
  case (Pair x y)
  note * = Pair [unfolded prod-set-simps]
  show ?case
    by (rule show-lawI)
      (auto simp del: o-apply intro!: o-append intro: show-lawD * simp: show-law-simps)
qed

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 \( \text{shows-string \ (string-of-digit \ n) } \)  
else \( \text{showsp-nat } p \ (n \ \text{div} \ 10) \) o \( \text{shows-string \ (string-of-digit \ (n \ \text{mod} \ 10))} \))

\text{declare } \text{showsp-nat\_simps} \ [\text{simp del}]

\text{lemma } \text{show-law-nat} \ [\text{show-law-intros}]:
\text{show-law \ showsp-nat } n
\text{ by (rule show-lawI, induct } n \text{ rule: nat-less-induct) (simp add: show-law-simps showsp-nat\_simps)}

\text{lemma } \text{showsp-nat-append} \ [\text{show-law-simps}]:
\text{showsp-nat } p \ n \ (x @ y) = \text{showsp-nat } p \ n \ x @ y
\text{ by (intro show-lawD show-law-intros)}

\text{definition } \text{showsp-int} :: \text{int showsp}
\text{ where}
\text{showsp-int } p \ i =
\ (if \( i < 0 \) then \( \text{shows-string } "{−}" \) o \text{showsp-nat } p \ (\text{nat } (− i)) \)  
else \text{showsp-nat } p \ (\text{nat } i))

\text{lemma } \text{show-law-int} \ [\text{show-law-intros}]:
\text{show-law \ showsp-int } i
\text{ by (rule show-lawI, cases } i < 0 \text{) (simp-all add: showsp-int\_def show-law-simps)}

\text{lemma } \text{showsp-int-append} \ [\text{show-law-simps}]:
\text{showsp-int } p \ i \ (x @ y) = \text{showsp-int } p \ i \ x @ y
\text{ by (intro show-lawD show-law-intros)}

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

\text{lemma } \text{show-law-rat} \ [\text{show-law-intros}]:
\text{show-law \ showsp-rat } r
\text{ by (rule show-lawI, cases quotient-of } r \text{) (simp add: showsp-rat\_def show-law-simps)}

\text{lemma } \text{showsp-rat-append} \ [\text{show-law-simps}]:
\text{showsp-rat } p \ r \ (x @ y) = \text{showsp-rat } p \ r \ x @ y
\text{ by (intro show-lawD show-law-intros)}

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

\text{local-setup} \ ⟨
\text{Show-Generator.register-foreign-partial-and-full-showsp @\{type-name prod\} 0}
\ @\{\text{term pshowsp-prod}\}

8
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