Haskell’s Show-Class in Isabelle/HOL*

Christian Sternagel René Thiemann

April 20, 2020

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 \( \mathbb{B}, \text{prod}, \text{sum}, \mathbb{N}, \mathbb{Z}, \) 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 7
   2.1 Displaying Polynomials .......................... 10

3 Show for Real Numbers – Interface 11

4 Show for Complex Numbers 12

5 Show Implementation for Real Numbers via Rational Numbers 13

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

*This research is supported by FWF (Austrian Science Fund) projects J3202 and P22767.
Deriving. Generator-Aux
Deriving. Derive-Manager
begin

type-synonym
  shows = string ⇒ string
— show-functions with precedence

- show-functions with precedence

- show-functions with precedence

1.1 The Show-Law

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

definition show-law :: 'a showsp ⇒ 'a ⇒ bool
where
  show-law s x ←→ (∀ p y z. s p x (y @ z) = s p x y @ z)

lemma show-lawI:
  (∀ 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 = (@)

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 \( \forall r s. \forall x \in \text{set} \ xs. \text{showsx} x (r @ s) = \text{showsx} x r \oplus s \)
  and \( \forall r s. \text{sep} (r @ s) = \text{sep} r \oplus s \)
  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 ⇒ '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 \( \forall r s. \forall x \in \text{set} \ xs. \text{showsx} x (r @ s) = \text{showsx} x r \oplus s \)
  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 pshowsp-list :: nat ⇒ shows list ⇒ shows
where
  pshowsp-list p xs = shows-list-gen id "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" 

3
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]:
showsp-list s p xs = shows-list-gen (s 0) "" "" "" "" xs
by (simp add: showsp-list-def pshowsp-list-def shows-list-gen-map)

lemma show-law-list [show-law-intros]:
(∀ x ∈ set xs ⇒ show-law s x) ⇒ show-law (showsp-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) = (#) c
definition shows-list (cs::string) = shows-string cs
instance
  by standard (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 (CHR 0x27) o s o shows (CHR 0x27)
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


\begin{verbatim}

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 standard (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)

\end{verbatim}
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 \( \forall r, s. \forall x \in \text{set} \ xs. \text{showx} \ x (r @ s) = \text{showx} \ x r @ s \)
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 \( \forall x \in \text{set} \ ys \Rightarrow f \ x = g \ x \)
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-all
qed simp

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

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

instantiation String.literal :: show
begin

definition shows-prec-literal :: nat \Rightarrow \text{String.literal} \Rightarrow \text{string} \Rightarrow \text{string}
where shows-prec p s = shows-string (String.explode s)

definition shows-list-literal :: \text{String.literal} list \Rightarrow \text{string} \Rightarrow \text{string}
where shows-list-list ss = shows-string (concat (map String.explode ss))

lemma shows-list-literal-code [code]:
shows-list = foldr (\( \lambda s. \text{shows-string} \ (\text{String.explode} \ s) \))
proof
  fix ss
\begin{verbatim}
show shows-list ss = foldr (\s. shows-string (String.explode s)) ss
  by (induct ss) (simp-all add: shows-list-literal-def shows-string-def)
qed

instance by standard
  (simp-all add: shows-prec-literal-def shows-list-literal-def shows-string-def)

end

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

\textbf{code-reserved} Haskell Show

end

2 Instances of the Show Class for Standard Types

theory Show-Instances
imports
  Show
  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)

abbreviation showsp-char :: char showsp
  where
    showsp-char \equiv shows-prec

lemma show-law-char [show-law-intros]:
  show-law showsp-char x
  by (rule show-lawI) (simp add: 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 pshowsprod :: (shows \times shows) showsp
\end{verbatim}
where
\[ p\text{showsp-prod} p (x, y) = \text{shows-string} "" o x o \text{shows-string } "", o y o \text{shows-string } "") \]

**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 1) (s2 1)

**lemma** showsp-prod-simps [simp, code]:
showsp-prod s1 s2 p (x, y) = shows-string "" o s1 1 x o shows-string "", o s2 1 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 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
by (rule show-lawI, induct n rule: nat-less-induct) (simp add: show-law-simps showsp-nat.simps)

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

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
by (rule show-lawI, cases i < 0) (simp-all add: show-law-simps showsp-int-def show-law-simps)

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

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
by (rule show-lawI, cases quotient-of r) (simp add: showsp-rat-def show-law-simps)

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

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.

local-setup
cshow-

Show-Generator.register-foreign-partial-and-full-showsp @{type-name prod} 0
@{term pshowsp-prod}
@{term showsp-prod} (SOME @{thm showsp-prod-def})
@{term map-prod} (SOME @{thm prod.map-comp}) [true, true]
@{thm show-law-prod}
#> Show-Generator.register-foreign-showsp @{typ unit} @{term showsp-unit}
@{thm show-law-unit}
#> Show-Generator.register-foreign-showsp @{typ bool} @{term showsp-bool}
derive show option sum prod unit bool nat int rat

default-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

end

2.1 Displaying Polynomials

We define a method which converts polynomials to strings and registers it in the Show class.

theory Show-Poly

imports
  Show-Instances
  HOL-Computational-Algebra.Polynomial

begin

fun show-factor :: nat ⇒ string where
  show-factor 0 = []
| show-factor (Suc 0) = "x"
| show-factor n = "x" ^ n @ show n

fun show-coeff-factor where
  show-coeff-factor c n = (if n = 0 then show c else if c = 1 then show-factor n
else show c @ show-factor n)

fun show-poly-main :: nat ⇒ 'a :: {zero,one,show} list ⇒ string where
  show-poly-main - [] = "0"
| show-poly-main n [c] = show-coeff-factor c n

10
show-poly-main \( n \ (c \# cs) = (\text{if } c = 0 \text{ then } \text{show-poly-main} \ (\text{Suc} \ n) \ cs \text{ else} \n\text{show-coeff-factor} \ c \ n \ @ \ " + \ " + \ @ \text{show-poly-main} \ (\text{Suc} \ n) \ cs) \)

definition show-poly :: 'a :: {zero,one,show}poly ⇒ string where
  show-poly \( p = \text{show-poly-main} \ 0 \ (\text{coeffs} \ p) \)

definition showsp-poly :: 'a :: {zero,one,show}poly showsp
where
  showsp-poly \( p \ x = \text{shows-string} \ (\text{show-poly} \ x) \)

instantiation poly :: ({show,one,zero}) show
begin

  definition shows-prec \( p \ (x :: 'a \ poly) = \text{showsp-poly} \ p \ x \)

  definition shows-list \( \ (ps :: 'a \ poly \ list) = \text{showsp-list} \ \text{shows-prec} \ 0 \ ps \)

lemma show-law-poly \[ \text{show-law-simps} \]:
  shows-prec \( p \ (a :: 'a \ poly) \ (r \ @ \ s) = \text{shows-prec} \ p \ a \ r \ @ \ s \)
  by (simp add: shows-prec-poly-def showsp-poly-def show-law-simps)

instance (by: standard (auto simp: shows-list-poly-def show-law-simps))

end
end

3 Show for Real Numbers – Interface

We just demand that there is some function from reals to string and register
this as show-function. Implementations are available in one of the theories
Show-Real-Impl and ../Algebraic-Numbers/Show-Real-.....

theory Show-Real
imports
  HOL.Real
  Show
begin

consts show-real :: real ⇒ string

definition showsp-real :: real showsp
where
  showsp-real \( p \ x \ y = \n(show-real \ x \ @ \ y) \)

lemma show-law-real \[ \text{show-law-intros} \]:
  show-law showsp-real \( r \)
  by (rule show-lawI) (simp add: showsp-real-def show-law-simps)

11
4 Show for Complex Numbers

We print complex numbers as real and imaginary parts. Note that by transitivity, this theory demands that an implementations for show-real is available, e.g., by using one of the theories Show-Real-Impl or ../Algebraic-Numbers/Show-Real-....
derive show complex
end

5 Show Implementation for Real Numbers via Rational Numbers

We just provide an implementation for show of real numbers where we assume that real numbers are implemented via rational numbers.

theory Show-Real-Impl
imports
  Show-Real
  Show-Instances
begin

  We now define show-real.
overloading show-real ≡ show-real
begin
definition show-real
  where show-real x ≡
    (if (∃ y. x = Ratreal y) then show (THE y. x = Ratreal y) else "Irrational")

end

lemma show-real-code[code]: show-real (Ratreal x) = show x
  unfolding show-real-def by auto
end

References