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 $\mathbb{B}$, $\text{prod}$, $\text{sum}$, $\mathbb{N}$, $\mathbb{Z}$, 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 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 $\Rightarrow$ string

   — show-functions with precedence
   type-synonym
      'a showsp = nat $\Rightarrow$ 'a $\Rightarrow$ 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.

\textbf{named-theorems} show-law-simps (simplification rules for proving the show law)
\textbf{named-theorems} show-law-intros (introduction rules for proving the show law)

\textbf{definition} show-law :: 'a showsp \Rightarrow 'a \Rightarrow bool
\text{where}
show-law \ s \ x \longleftrightarrow (\forall p \ y \ z. \ s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z)

\textbf{lemma} show-lawI:
(show-law \ s \ x \Longrightarrow (s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z) \Longrightarrow \text{show-law } s \ x)
\text{by (simp add: show-law-def)}

\textbf{lemma} show-lawE:
(show-law \ s \ x \Longrightarrow (s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z) \Longrightarrow P) \Longrightarrow P
\text{by (auto simp: show-law-def)}

\textbf{lemma} show-lawD:
show-law \ s \ x \Longrightarrow s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z
\text{by (blast elim: show-lawE)}

\textbf{class} show =
\text{fixes} shows-prec :: 'a showsp
\text{and} shows-list :: 'a list \Rightarrow shows
\text{assumes} shows-prec-append [show-law-simps]: shows-prec \ p \ x \ (r \ @ \ s) = shows-prec \ p \ x \ r \ @ \ s
\text{and}
shows-list-append [show-law-simps]: shows-list \ xs \ (r \ @ \ s) = shows-list \ xs \ r \ @ \ s
\text{begin}
\textbf{abbreviation} shows \ x \equiv \text{shows-prec } 0 \ x
\textbf{abbreviation} show \ x \equiv shows \ x ""
\text{end}

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

\textbf{definition} shows-string :: string \Rightarrow shows
\text{where}
shows-string = \ op \ @

\textbf{lemma} shows-string-append [show-law-simps]:
shows-string \ x \ (r \ @ \ s) = shows-string \ x \ r \ @ \ s
\text{by (simp add: shows-string-def)}

\textbf{fun} shows-sep :: ('a \Rightarrow shows) \Rightarrow shows \Rightarrow 'a list \Rightarrow shows
\text{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** \(\forall r s. \forall x \in \text{set} \, xs. \text{shows} \, x \, (r \oplus s) = \text{shows} \, x \, r \oplus s\)
and \(\forall r s. \text{sep} \, (r \oplus s) = \text{sep} \, r \oplus s\)
**shows** shows-sep shows-sep sep xs (r \oplus s) = shows-sep shows-sep sep xs r \oplus 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 \Rightarrow \text{shows}) \Rightarrow \text{string} \Rightarrow \text{string} \Rightarrow \text{string} \Rightarrow \text{string} \Rightarrow \text{'a list} 
\Rightarrow \text{shows}

where
shows-list-gen showsx e l s r xs =
  (if xs \in \text{[\]} \ then shows-string e
  else shows-string l o shows-sep showsx \(\text{shows-string} \, s\) \(\text{xs} \, o \, \text{shows-string} \, r\))

**lemma** shows-list-gen-append [show-law-simps]:
**assumes** \(\forall r s. \forall x \in \text{set} \, xs. \text{shows} \, x \, (r \oplus s) = \text{shows} \, x \, r \oplus s\)
**shows** shows-list-gen showsx e l sep r xs (s \oplus t) = shows-list-gen showsx e l sep r xs s \oplus 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 \Rightarrow \text{shows list} \Rightarrow \text{shows}

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

**definition** showsp-list :: 'a showsp \Rightarrow nat \Rightarrow \text{'a list} \Rightarrow \text{shows}

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

**lemma** showsp-list-code [code]:
shows-list p xs = shows-list-gen (s 0) "[]" "[]" "[]" "[]" xs
by (simp add: showsp-list-code 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} \, (\text{shows-list} \, s) \, \text{xs}
by (simp add: show-law-def showsp-list-code show-law-simps)
lemma shows-list-append [show-law-simps]:
\[ (\forall p y z. \forall x \in \text{set } xs. \ s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z) \implies \]
shows-list s p xs (y @ z) = shows-list s p xs y @ z
by (simp add: show-law-simps shows-list-def pshows-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 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 (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]: shows-paren s (x @ y) = shows-paren s x @ y
and
shows-quote-append [show-law-simps]: 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:
\[ (\forall x y. f \ (x \ @ \ y) = f \ x \ @ \ y) \implies g \ (x \ @ \ y) = g \ x \ @ \ y \implies (f \ o \ g) \ (x \ @ \ y) = (f \ o \ g) \ x \ @ \ y \]
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)

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 ∀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]:
\begin{align*}
\text{assumes } & xs = ys \text{ and } \prod x. x \in \text{set } ys \implies f x = g x \\
\text{shows } & \text{shows-sep } f \text{ sep } xs = \text{shows-sep } g \text{ sep } ys \\
\text{using } & \text{assms} \\
\text{proof } & (\text{induct } ys \text{ arbitrary: } xs) \\
& \text{case } (\text{Cons } y \text{ ys}) \\
& \text{then show } \text{?case by } (\text{cases } ys) \text{ simp-all} \\
\text{qed simp}
\end{align*}

\text{lemma shows-list-gen-cong [fundef-cong]:} \\
\text{assumes } xs = ys \text{ and } \prod x. x \in \text{set } ys \implies f x = g x \\
\text{shows } \text{shows-list-gen } f e l \text{ sep } r \text{ xs} = \text{shows-list-gen } g e l \text{ sep } r \text{ ys} \\
\text{using } \text{shows-sep-cong [of } xs \text{ ys } f \text{ g] assms by } (\text{cases } xs) \text{ (auto simp: shows-list-gen-def)}

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

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

\text{abbreviation (input) shows-append :: shows } \Rightarrow \text{ shows } \Rightarrow \text{ shows } (\text{infixr } +@ + 10) \\
\text{where} \\
\text{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.

\text{code-reserved Haskell Show}

end

\section{Instances of the Show Class for Standard Types}

\text{theory Show-Instances} \\
\text{imports} \\
\text{Show} \\
\text{~/src/HOL/Rat} \\
\text{begin}

\text{definition showsp-unit :: unit showsp} \\
\text{where} \\
\text{showsp-unit } p \text{ x} = \text{shows-string } "()"

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

\text{abbreviation showsp-char :: char showsp}
where
shows-char ≡ shows-prec

lemma show-law-char [show-law-intros]:
show-law shows-char x
by (rule show-lawI) (simp add: show-law-simps)

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

lemma show-law-bool [show-law-intros]:
show-law shows-bool x
by (rule show-lawI, cases x) (simp-all add: show-law-simps)

primrec pshows-prod :: (shows × shows) showsp
where
pshows-prod p (x, y) = shows-string "" o s1 0 x o shows-string "," o s2 0 y o shows-string ""

definition shows-prod :: 'a showsp ⇒ 'b showsp ⇒ ('a × 'b) showsp
where
[code del]: shows-prod s1 s2 p = pshows-prod p o map-prod (s1 0) (s2 0)

lemma shows-prod-simps [simp, code]:
shows-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: shows-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 (shows-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")

7
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: 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) \Rightarrow
    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]:
\[
\text{shows-rat } p \ r \ (x \ @ \ y) = \text{shows-rat } p \ r \ x \ @ \ y
\]

by \ (\text{intro show-lawD show-law-intros})

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

\texttt{local-setup} \langle\langle
\begin{align*}
\text{Show-Generator.register-foreign-partial-and-full-showsp } & \{\text{type-name prod}\} \ 0 \\
& \{\text{term pshowsp-prod}\} \\
& \{\text{term showsp-prod}\} \ (\text{SOME } \{\text{thm showsp-prod-def}\}) \\
& \{\text{term map-prod}\} \ (\text{SOME } \{\text{thm prod.map-comp}\} \ [true, true]) \\
& \{\text{thm show-law-prod}\} \\
& \#> \text{Show-Generator.register-foreign-showsp } \{\text{typ unit}\} \ {\{\text{term showsp-unit}\}} \\
& \{\text{thm show-law-unit}\} \\
& \#> \text{Show-Generator.register-foreign-showsp } \{\text{typ bool}\} \ {\{\text{term showsp-bool}\}} \\
& \{\text{thm show-law-bool}\} \\
& \#> \text{Show-Generator.register-foreign-showsp } \{\text{typ char}\} \ {\{\text{term showsp-char}\}} \\
& \{\text{thm show-law-char}\} \\
& \#> \text{Show-Generator.register-foreign-showsp } \{\text{typ nat}\} \ {\{\text{term showsp-nat}\}} \\
& \{\text{thm show-law-nat}\} \\
& \#> \text{Show-Generator.register-foreign-showsp } \{\text{typ int}\} \ {\{\text{term showsp-int}\}} \ {\{\text{thm show-law-int}\}} \\
& \#> \text{Show-Generator.register-foreign-showsp } \{\text{typ rat}\} \ {\{\text{term showsp-rat}\}} \ {\{\text{thm show-law-rat}\}}
\end{align*}
\rangle
\rangle

\texttt{derive} \textit{show} option sum prod unit bool nat int rat

\texttt{export-code}
\begin{align*}
\text{shows-prec} \ & :: \ 'a::show \ option \ showsp \\
\text{shows-prec} \ & :: \ (\text{\'a::show}, \ \text{\'b::show}) \ \text{sum} \ showsp \\
\text{shows-prec} \ & :: \ (\text{\'a::show} \ \times \ \text{\'b::show}) \ showsp \\
\text{shows-prec} \ & :: \ \text{unit} \ showsp \\
\text{shows-prec} \ & :: \ \text{char} \ showsp \\
\text{shows-prec} \ & :: \ \text{bool} \ showsp \\
\text{shows-prec} \ & :: \ \text{nat} \ showsp \\
\text{shows-prec} \ & :: \ \text{int} \ showsp \\
\text{shows-prec} \ & :: \ \text{rat} \ showsp
\end{align*}

\texttt{checking}

end

References