Haskell’s Show-Class in Isabelle/HOL*

Christian Sternagel        René Thiemann

August 16, 2018

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 ℂ, 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
   2.1 Displaying Polynomials .............................. 9
3 Show for Real Numbers – Interface 10
4 Show for Complex Numbers 11
5 Show Implementetion for Real Numbers via Rational Numbers 11

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

type-synonym
  'a showsp = nat ⇒ 'a ⇒ shows

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.

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
  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
  ⟨proof⟩

lemma show-lawE:
  show-law s x ⇒ (s p x (y @ z) = s p x y @ z ⇒ P) ⇒ P
  ⟨proof⟩

lemma show-lawD:
  show-law s x ⇒ s p x (y @ z) = s p x y @ z
  ⟨proof⟩

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

⟨proof⟩

**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

∀ 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 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

∀ 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 ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′ ′′  

3
lemma showsp-list-code [code]:
  showsp-list s p xs = shows-list-gen (s 0) "|" "|"静静 "|"" xs
 ⟨proof⟩

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

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
 ⟨proof⟩

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 ⟨proof⟩

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]: shows-space (x @ y) = shows-space (x @ y)
  and
  shows-quote-append [show-law-simps]: shows-space (x @ y) = shows-space (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⟩
\textbf{lemma} \textit{o-append}:
\[
(\forall x \ y. \ f \ (x \ @ \ y) = f \ x \ @ \ y) \implies g \ (x \ @ \ y) = g \ x \ @ \ y \implies (f \ o \ g) \ (x \ @ \ y) = (f \ o \ g) \ x \ @ \ y
\]
\langle proof \rangle

\textbf{instantiation list :: (show) show}
begin

\textbf{definition} shows-prec \ (p :: nat) \ (xs :: 'a list) = shows-list xs
\textbf{definition} shows-list \ (xs :: 'a list list) = showsp-list shows-prec 0 xs

\textbf{instance}
\langle proof \rangle
end

\textbf{definition} shows-lines :: 'a::show list \Rightarrow shows
\textbf{where}
shows-lines = shows-sep shows shows-nl
\textbf{definition} shows-many :: 'a::show list \Rightarrow shows
\textbf{where}
shows-many = shows-sep shows id
\textbf{definition} shows-words :: 'a::show list \Rightarrow shows
\textbf{where}
shows-words = shows-sep shows shows-space

\textbf{lemma} shows-lines-append [show-law-simps]:
\[\text{shows-lines } xs \ (r \ @ \ s) = \text{shows-lines } xs \ r \ @ \ s\]
\langle proof \rangle

\textbf{lemma} shows-many-append [show-law-simps]:
\[\text{shows-many } xs \ (r \ @ \ s) = \text{shows-many } xs \ r \ @ \ s\]
\langle proof \rangle

\textbf{lemma} shows-words-append [show-law-simps]:
\[\text{shows-words } xs \ (r \ @ \ s) = \text{shows-words } xs \ r \ @ \ s\]
\langle proof \rangle

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

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

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

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

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

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

\textbf{code-reserved} Haskell Show

end

2 Instances of the Show Class for Standard Types

theory Show-Instances
imports  
Show  
HOL.Rat
begin

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

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

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

\textbf{lemma} show-law-char [show-law-intros]:  
show-law showsp-char \ x
primrec shows-bool :: bool showsp
where
  shows-bool p True = shows-string "True" |
  shows-bool p False = shows-string "False"

lemma show-law-bool [show-law-intros]:
  show-law shows-bool x

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 showsprod :: 'a showsp ⇒ 'b showsp ⇒ ('a × 'b) showsp
where
  [code del]: showsprod s1 s2 p = pshowsp-prod p o map-prod (s1 1) (s2 1)

lemma showsprod-simps [simp, code]:
  showsprod s1 s2 p (x, y) =
  shows-string "(" o s1 1 x o shows-string ", " o s2 1 y o shows-string ")"

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 (showsprod s1 s2) y

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 shows-nat :: nat showsp
where
  shows-nat p n =
(if \( n < 10 \) then \( \text{shows-string (string-of-digit n)} \)
else \( \text{shows-nat p (n div 10)} \) o \( \text{shows-string (string-of-digit (n mod 10))} \))

**declare** shows-nat.simps [simp del]

**lemma** show-law-nat [show-law-intros]:
\( \text{show-law shows-nat n} \)
\( \langle \text{proof} \rangle \)

**lemma** shows-nat-append [show-law-simps]:
\( \text{shows-nat p n (x @ y)} = \text{shows-nat p n x @ y} \)
\( \langle \text{proof} \rangle \)

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

**lemma** show-law-int [show-law-intros]:
\( \text{show-law showsp-int i} \)
\( \langle \text{proof} \rangle \)

**lemma** showsp-int-append [show-law-simps]:
\( \text{shows-p-int p i (x @ y)} = \text{shows-nat p i x @ y} \)
\( \langle \text{proof} \rangle \)

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

**lemma** show-law-rat [show-law-intros]:
\( \text{show-law showsp-rat r} \)
\( \langle \text{proof} \rangle \)

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

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 "\text{Unity}" and "\text{Pair}"; for \textit{nat}, we do not want to display "\text{Suc (Suc (Suc 0 ...))}"; and neither \textit{int} nor \textit{rat} are datatypes.

\( \langle \text{ML} \rangle \)

**derive** show option sum prod unit bool nat int rat
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" ^ @ 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 [c] = show-coeff-factor c n
| show-poly-main (c # cs) = (if c = 0 then show-poly-main (Suc n) cs else
  show-coeff-factor c n @ "+" @ show-poly-main (Suc n) cs)

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

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

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

end
**definition** shows-prec p (x :: 'a poly) = showsp-poly p x

**definition** shows-list (ps :: 'a poly list) = showsp-list shows-prec 0 ps

**lemma** show-law-poly [show-law-simps]:

shows-prec p (a :: 'a poly) (r @ s) = shows-prec p a r @ s

⟨proof⟩

instance ⟨proof⟩

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 =
(show-real x @ y)

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

show-law showsp-real r
⟨proof⟩

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

showsp-real p r (x @ y) = showsp-real p r x @ y
⟨proof⟩

⟨ML⟩

derive show real

end
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-....

theory Show-Complex
imports
  HOL.Complex
  Show-Real
begin

definition show-complex x = (let r = Re x; i = Im x in if (i = 0) then show-real r else if r = 0 then show-real i @ "i" else "(" @ show-real r @ "+" @ show-real i @ ")")
definition showsp-complex :: complex showsp
where
  showsp-complex p x y = (show-complex x @ y)

lemma show-law-complex [show-law-intros]:
  show-law showsp-complex r
  ⟨proof⟩

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

⟨ML⟩
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 \textit{show-real}.

\begin{verbatim}
overloading show-real \equiv show-real
begin
  definition show-real
  where show-real x \equiv
      (if (\exists y. x = \text{Ratreal } y) then show (THE y. x = \text{Ratreal } y) else "Irrational")
end

lemma show-real-code[\textit{code}]: show-real (\text{Ratreal } x) = show x
  (proof)
end
\end{verbatim}

References


12