

Haskell’s **Show**-Class in Isabelle/HOL*

Christian Sternagel René Thiemann

December 14, 2021

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
2.1	Displaying Polynomials	9
3	Show for Real Numbers – Interface	10
4	Show for Complex Numbers	11
5	Show Implemetation for Real Numbers via Rational Numbers	12

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 \Rightarrow *string*

— show-functions with precedence

type-synonym
'a showsp = nat \Rightarrow *'a* \Rightarrow *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* \langle *simplification rules for proving the show law* \rangle

named-theorems *show-law-intros* \langle *introduction rules for proving the show law* \rangle

definition *show-law* :: *'a showsp* \Rightarrow *'a* \Rightarrow *bool*

where

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

lemma *show-lawI*:

$(\bigwedge p y z. s p x (y @ z) = s p x y @ z) \Longrightarrow show-law s x$
 \langle *proof* \rangle

lemma *show-lawE*:

show-law s x $\Longrightarrow (s p x (y @ z) = s p x y @ z \Longrightarrow P) \Longrightarrow P$
 \langle *proof* \rangle

lemma *show-lawD*:

show-law s x $\Longrightarrow s p x (y @ z) = s p x y @ z$
 \langle *proof* \rangle

class *show* =

fixes *shows-prec* :: *'a showsp*

and *shows-list* :: *'a list* \Rightarrow *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* \equiv *shows-prec 0 x*

abbreviation *show x* \equiv *shows x* *''''*

end

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

definition *shows-string* :: *string* \Rightarrow *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* \Rightarrow *shows*) \Rightarrow *shows* \Rightarrow '*a* *list* \Rightarrow *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 $\bigwedge r s. \forall x \in \text{set } xs. \text{shows } x (r @ s) = \text{shows } x r @ s$

and $\bigwedge r s. \text{sep } (r @ s) = \text{sep } r @ s$

shows *shows-sep* *shows* *x* *sep* *xs* (*r* @ *s*) = *shows-sep* *shows* *x* *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* \Rightarrow *shows*) \Rightarrow *string* \Rightarrow *string* \Rightarrow *string* \Rightarrow *string* \Rightarrow '*a* *list* \Rightarrow *shows*

where

shows-list-gen *shows* *x* *e* *l* *s* *r* *xs* =

(if *xs* = [] then *shows-string* *e*

else *shows-string* *l* *o shows-sep* *shows* *x* (*shows-string* *s*) *xs* *o shows-string* *r*)

lemma *shows-list-gen-append* [*show-law-simps*]:

assumes $\bigwedge r s. \forall x \in \text{set } xs. \text{shows } x (r @ s) = \text{shows } x r @ s$

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

where

pshowsp-list *p* *xs* = *shows-list-gen* *id* "" "" ["", ""], "" "" *xs*

definition *showsp-list* :: '*a* *showsp* \Rightarrow *nat* \Rightarrow '*a* *list* \Rightarrow *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
 ⟨*proof*⟩

lemma *show-law-list* [*show-law-intros*]:
 $(\bigwedge x. x \in \text{set } xs \implies \text{show-law } s \ x) \implies \text{show-law } (\text{showsp-list } s) \ xs$
 ⟨*proof*⟩

lemma *showsp-list-append* [*show-law-simps*]:
 $(\bigwedge p \ y \ z. \forall x \in \text{set } xs. s \ p \ x \ (y \ @ \ z) = s \ p \ x \ y \ @ \ z) \implies$
 $\text{showsp-list } s \ p \ xs \ (y \ @ \ z) = \text{showsp-list } s \ p \ xs \ y \ @ \ z$
 ⟨*proof*⟩

1.2 Show-Functions for Characters and Strings

instantiation *char* :: *show*
begin

definition *shows-prec* $p \ (c::\text{char}) = (\#) \ c$
definition *shows-list* $(cs::\text{string}) = \text{shows-string } cs$
instance
 ⟨*proof*⟩

end

definition *shows-nl* = *shows* (CHR "↵")
definition *shows-space* = *shows* (CHR " ")
definition *shows-paren* $s = \text{shows } (CHR "(") \ o \ s \ o \ \text{shows } (CHR ")")$
definition *shows-quote* $s = \text{shows } (CHR "0x27") \ o \ s \ o \ \text{shows } (CHR "0x27")$
abbreviation *apply-if* $b \ s \equiv (\text{if } b \ \text{then } s \ \text{else } \text{id})$ — conditional function application

Parenthesize only if precedence is greater than 0.

definition *shows-pl* $(p::\text{nat}) = \text{apply-if } (p > 0) \ (\text{shows } (CHR "("))$
definition *shows-pr* $(p::\text{nat}) = \text{apply-if } (p > 0) \ (\text{shows } (CHR ")")$

lemma
shows-nl-append [*show-law-simps*]: *shows-nl* $(x \ @ \ y) = \text{shows-nl } x \ @ \ y$ **and**
shows-space-append [*show-law-simps*]: *shows-space* $(x \ @ \ y) = \text{shows-space } x \ @ \ y$
and
shows-paren-append [*show-law-simps*]:
 $(\bigwedge x \ y. s \ (x \ @ \ y) = s \ x \ @ \ y) \implies \text{shows-paren } s \ (x \ @ \ y) = \text{shows-paren } s \ x \ @ \ y$ **and**
shows-quote-append [*show-law-simps*]:
 $(\bigwedge x \ y. s \ (x \ @ \ y) = s \ x \ @ \ y) \implies \text{shows-quote } s \ (x \ @ \ y) = \text{shows-quote } s \ x \ @ \ y$
and
shows-pl-append [*show-law-simps*]: *shows-pl* $p \ (x \ @ \ y) = \text{shows-pl } p \ x \ @ \ y$ **and**
shows-pr-append [*show-law-simps*]: *shows-pr* $p \ (x \ @ \ y) = \text{shows-pr } p \ x \ @ \ y$
 ⟨*proof*⟩

lemma *o-append*:

$(\bigwedge x y. f (x @ y) = f x @ y) \implies g (x @ y) = g x @ y \implies (f o g) (x @ y) = (f o g) x @ y$
<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* \Rightarrow *shows*

where

shows-lines = *shows-sep shows shows-nl*

definition *shows-many* :: 'a::*show list* \Rightarrow *shows*

where

shows-many = *shows-sep shows id*

definition *shows-words* :: 'a::*show list* \Rightarrow *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 $\bigwedge r s. \forall x \in \text{set } xs. \text{showx } x (r @ s) = \text{showx } x r @ s$
shows *foldr showx xs (r @ s) = foldr showx xs r @ s*
<proof>

lemma *shows-sep-cong* [*fundef-cong*]:

assumes *xs = ys* **and** $\bigwedge x. x \in \text{set } ys \implies f x = g x$

shows *shows-sep f sep xs = shows-sep g sep ys*
<proof>

lemma *shows-list-gen-cong [fundef-cong]:*
assumes *xs = ys and $\bigwedge x. x \in \text{set } ys \implies f x = g x$*
shows *shows-list-gen f e l sep r xs = shows-list-gen g e l sep r ys*
<proof>

lemma *showsp-list-cong [fundef-cong]:*
xs = ys $\implies p = q \implies$
($\bigwedge p x. x \in \text{set } ys \implies f p x = g p x$) \implies showsp-list f p xs = showsp-list g q ys
<proof>

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

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

instantiation *String.literal :: show*
begin

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

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

lemma *shows-list-literal-code [code]:*
shows-list = foldr ($\lambda s. \text{shows-string (String.explode s)}$)
<proof>

instance <proof>

end

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

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*
 ⟨*proof*⟩

abbreviation *showsp-char* :: *char showsp*
where
 showsp-char ≡ *shows-prec*

lemma *show-law-char* [*show-law-intros*]:
 show-law *showsp-char* *x*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

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* 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* ")"
 ⟨*proof*⟩

lemma *show-law-prod* [*show-law-intros*]:
 ($\bigwedge x. x \in \text{Basic-BNFs.fsts } y \implies \text{show-law } s1 \ x \implies$
 ($\bigwedge x. x \in \text{Basic-BNFs.snds } y \implies \text{show-law } s2 \ x \implies$
 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 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
 ⟨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

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^" @ 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
| show-poly-main n (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
begin

```

```

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 $r = \text{Re } x; i = \text{Im } x$ *in*

if $(i = 0)$ *then* *show-real* r *else if*

$r = 0$ *then* *show-real* $i @ "i"$ *else*

$"(" @ \text{show-real } r @ "+" @ \text{show-real } i @ "i)"$

definition *showsp-complex* $:: \text{complex showsp}$

where

showsp-complex $p x y =$

$(\text{show-complex } x @ y)$

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 *show-real*.

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*[code]: *show-real* (Ratreal x) = show x

<proof>

end

References

- [1] P. Hudak, J. Peterson, and J. H. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5), 1992. Original version at <http://doi.acm.org/10.1145/130697.130698>, updated version at <https://www.haskell.org/tutorial/>.