

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

Christian Sternagel René Thiemann

February 6, 2026

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	7
2.1	Displaying Polynomials	9
3	Show Based on String Literals	10
4	Show for Real Numbers – Interface	14
5	Show for Complex Numbers	15
6	Show Implemetation for Real Numbers via Rational Numbers	15

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*

*This research is supported by FWF (Austrian Science Fund) projects J3202 and P22767.

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

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* \Rightarrow *shows*

where

$$\text{shows-string} = (@)$$

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

$$\text{shows-string } x (r @ s) = \text{shows-string } x r @ s$$

<proof>

fun *shows-sep* :: ('*a* \Rightarrow *shows*) \Rightarrow *shows* \Rightarrow '*a list* \Rightarrow *shows*

where

$$\text{shows-sep } s \text{ sep } [] = \text{shows-string } "" |$$

$$\text{shows-sep } s \text{ sep } [x] = s x |$$

$$\text{shows-sep } s \text{ sep } (x\#xs) = s x o \text{ sep } o \text{ shows-sep } s \text{ sep } xs$$

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

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

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

$$\text{shows } \text{shows-sep } \text{shows } x \text{ sep } xs (r @ s) = \text{shows-sep } \text{shows } x \text{ sep } xs r @ s$$

<proof>

lemma *shows-sep-map*:

$$\text{shows-sep } f \text{ sep } (\text{map } g \text{ } xs) = \text{shows-sep } (f o g) \text{ sep } xs$$

<proof>

definition

shows-list-gen :: ('*a* \Rightarrow *shows*) \Rightarrow *string* \Rightarrow *string* \Rightarrow *string* \Rightarrow *string* \Rightarrow '*a list* \Rightarrow *shows*

where

$$\text{shows-list-gen } \text{shows } x e l s r xs =$$

$$(\text{if } xs = [] \text{ then } \text{shows-string } e$$

$$\text{else } \text{shows-string } l o \text{ shows-sep } \text{shows } x (\text{shows-string } s) xs o \text{ shows-string } r)$$

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

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

$$\text{shows } \text{shows-list-gen } \text{shows } x e l \text{ sep } r xs (s @ t) = \text{shows-list-gen } \text{shows } x e l \text{ sep } r xs s @ t$$

<proof>

lemma *shows-list-gen-map*:

$$\text{shows-list-gen } f e l \text{ sep } r (\text{map } g \text{ } xs) = \text{shows-list-gen } (f o g) e l \text{ sep } r xs$$

<proof>

definition *pshowsp-list* :: *nat* \Rightarrow *shows list* \Rightarrow *shows*

where

$$\text{pshowsp-list } p \text{ } xs = \text{shows-list-gen } \text{id } "" "" [" ", " "] xs$$

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

<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  $\text{shows-sep } f \text{ sep } xs = \text{shows-sep } g \text{ sep } ys$ 
  <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  $\text{shows-list-gen } f \text{ e l sep } r \text{ } xs = \text{shows-list-gen } g \text{ e l sep } r \text{ } ys$ 
  <proof>

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

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

abbreviation (input) shows-append :: shows  $\Rightarrow$  shows  $\Rightarrow$  shows (infixr <+@+>
10)
where
   $s \text{ +@+ } p \equiv s \circ p$ 

instantiation String.literal :: show
begin

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

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

lemma shows-list-literal-code [code]:
   $\text{shows-list} = \text{foldr } (\lambda s. \text{shows-string } (\text{String.explode } s))$ 
  <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 \equiv *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* \times *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* \Rightarrow '*b* *showsp* \Rightarrow ('*a* \times '*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*]:

$(\bigwedge x. x \in \text{Basic-BNFs.fsts } y \implies \text{show-law } s1 \ x) \implies$
 $(\bigwedge x. x \in \text{Basic-BNFs.snds } y \implies \text{show-law } s2 \ x) \implies$
 $\text{show-law } (\text{showsp-prod } s1 \ s2) \ y$
 <proof>

definition *string-of-digit* :: nat \Rightarrow 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* $p \ (n \text{ div } 10)$ o *shows-string* (*string-of-digit* ($n \text{ mod } 10$)))

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 $p \ n \ (x \ @ \ y) = \text{showsp-nat } p \ n \ x \ @ \ y$
 <proof>

definition *showsp-int* :: int *showsp*

where

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

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

show-law *showsp-int* i
 <proof>

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

showsp-int $p \ i \ (x \ @ \ y) = \text{showsp-int } p \ i \ x \ @ \ y$
 <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 Based on String Literals

theory Shows-Literal

imports

Main

Show-Instances

begin

In this theory we provide an alternative to the *show*-class, where *String.literal* instead of *string* is used, with the aim that target-language readable strings are used in generated code. In particular when writing Isabelle functions that produce strings such as *STR "this is info for the user: ..."*, this class

might be useful.

To keep it simple, in contrast to *show*, here we do not enforce the show law.

type-synonym *showsl* = *String.literal* \Rightarrow *String.literal*

definition *showsl-of-shows* :: *shows* \Rightarrow *showsl* **where**
showsl-of-shows *shws* *s* = *String.implode* (*shws* []) + *s*

definition *showsl-lit* :: *String.literal* \Rightarrow *showsl* **where**
showsl-lit = (+)

definition *showsl-paren* *s* = *showsl-lit* (*STR* "(") *o* *s* *o* *showsl-lit* (*STR* ")")

fun *showsl-sep* :: ('*a* \Rightarrow *showsl*) \Rightarrow *showsl* \Rightarrow '*a* list \Rightarrow *showsl*
where
showsl-sep *s* *sep* [] = *showsl-lit* (*STR* "'") |
showsl-sep *s* *sep* [*x*] = *s* *x* |
showsl-sep *s* *sep* (*x*#*xs*) = *s* *x* *o* *sep* *o* *showsl-sep* *s* *sep* *xs*

definition
showsl-list-gen :: ('*a* \Rightarrow *showsl*) \Rightarrow *String.literal* \Rightarrow *String.literal*
 \Rightarrow *String.literal* \Rightarrow *String.literal* \Rightarrow '*a* list \Rightarrow *showsl*
where
showsl-list-gen *showslx* *e* *l* *s* *r* *xs* =
(if *xs* = [] then *showsl-lit* *e*
else *showsl-lit* *l* *o* *showsl-sep* *showslx* (*showsl-lit* *s*) *xs* *o* *showsl-lit* *r*)

definition *default-showsl-list* :: ('*a* \Rightarrow *showsl*) \Rightarrow '*a* list \Rightarrow *showsl* **where**
default-showsl-list *sl* = *showsl-list-gen* *sl* (*STR* "[]") (*STR* "'") (*STR* ", ") (*STR* "']")

definition [code-unfold]: *char-zero* = (48 :: integer)

lemma *char-zero*: *char-zero* = *integer-of-char* (*CHR* "0") \langle proof \rangle

fun *lit-of-digit* :: nat \Rightarrow *String.literal* **where**
lit-of-digit *n* =
String.implode [*char-of-integer* (*char-zero* + *integer-of-nat* *n*)]

class *showl* =
fixes *showsl* :: '*a* \Rightarrow *showsl*
and *showsl-list* :: '*a* list \Rightarrow *showsl*

definition *showsl-lines* *desc-empty* = *showsl-list-gen* *showsl* *desc-empty* (*STR* """)
(*STR* "↔") (*STR* """)

abbreviation *showl* **where** *showl* *x* \equiv *showsl* *x* (*STR* """)

instantiation *char* :: *showl*

```

begin
definition showsl-char  $c = \text{showsl-lit } (\text{String.implode } [c])$  — Shouldn't there be a
faster conversion than via strings?
definition showsl-list-char  $cs\ s = \text{showsl-lit } (\text{String.implode } cs)\ s$ 
instance  $\langle \text{proof} \rangle$ 
end

instantiation String.literal :: showl
begin
definition showsl ( $s :: \text{String.literal}$ ) = showsl-lit  $s$ 
definition showsl-list ( $xs :: \text{String.literal list}$ ) = default-showsl-list showsl  $xs$ 
instance  $\langle \text{proof} \rangle$ 
end

instantiation bool :: showl
begin
definition showsl ( $b :: \text{bool}$ ) = showsl-lit (if  $b$  then STR "True" else STR "False")

definition showsl-list ( $xs :: \text{bool list}$ ) = default-showsl-list showsl  $xs$ 
instance  $\langle \text{proof} \rangle$ 
end

instantiation nat :: showl
begin
fun showsl-nat :: nat  $\Rightarrow$  showsl where
  showsl-nat  $n =$ 
    (if  $n < 10$  then showsl-lit (lit-of-digit  $n$ )
    else showsl-nat ( $n \text{ div } 10$ ) o showsl-lit (lit-of-digit ( $n \text{ mod } 10$ )))
definition showsl-list ( $xs :: \text{nat list}$ ) = default-showsl-list showsl  $xs$ 
instance  $\langle \text{proof} \rangle$ 
end

instantiation int :: showl
begin
definition showsl-int  $i =$ 
  (if  $i < 0$  then showsl-lit (STR "-" ) o showsl (nat ( $- i$ )) else showsl (nat  $i$ ))
definition showsl-list ( $xs :: \text{int list}$ ) = default-showsl-list showsl  $xs$ 
instance  $\langle \text{proof} \rangle$ 
end

instantiation integer :: showl
begin
definition showsl-integer :: integer  $\Rightarrow$  showsl where showsl-integer  $i = \text{showsl}$ 
(int-of-integer  $i$ )
definition showsl-list-integer :: integer list  $\Rightarrow$  showsl where showsl-list-integer  $xs$ 
= default-showsl-list showsl  $xs$ 
instance  $\langle \text{proof} \rangle$ 
end

```

```

instantiation rat :: showl
begin
definition showsl-rat x =
  (case quotient-of x of (d, n) =>
    if n = 1 then showsl d else showsl d o showsl-lit (STR "'/'") o showsl n)
definition showsl-list (xs :: rat list) = default-showsl-list showsl xs
instance <proof>
end

instantiation unit :: showl
begin
definition showsl (x :: unit) = showsl-lit (STR "()")
definition showsl-list (xs :: unit list) = default-showsl-list showsl xs
instance <proof>
end

instantiation option :: (showl) showl
begin
fun showsl-option where
  showsl-option None = showsl-lit (STR "None")
| showsl-option (Some x) = showsl-lit (STR "Some ('') o showsl x o showsl-lit (STR ''')")
definition showsl-list (xs :: 'a option list) = default-showsl-list showsl xs
instance <proof>
end

instantiation sum :: (showl,showl) showl
begin
fun showsl-sum where
  showsl-sum (Inl x) = showsl-lit (STR "Inl ('') o showsl x o showsl-lit (STR '')")
| showsl-sum (Inr x) = showsl-lit (STR "Inr ('') o showsl x o showsl-lit (STR '')")
definition showsl-list (xs :: ('a + 'b) list) = default-showsl-list showsl xs
instance <proof>
end

instantiation prod :: (showl,showl) showl
begin
fun showsl-prod where
  showsl-prod (Pair x y) = showsl-lit (STR "('') o showsl x
    o showsl-lit (STR ', '') o showsl y o showsl-lit (STR '')")
definition showsl-list (xs :: ('a * 'b) list) = default-showsl-list showsl xs
instance <proof>
end

definition [code-unfold]: showsl-nl = showsl (STR "[←]")

```

definition *add-index* :: *showsl* \Rightarrow *nat* \Rightarrow *showsl* **where**
add-index *s i* = *s* *o* *showsl-lit* (*STR* *''*) *o* *showsl* *i*

instantiation *list* :: (*showl*) *showl*

begin

definition *showsl-list* :: '*a list* \Rightarrow *showsl* **where**

showsl-list (*xs* :: '*a list*) = *showl-class.showsl-list* *xs*

definition *showsl-list-list* (*xs* :: '*a list list*) = *default-showsl-list* *showsl* *xs*

instance \langle *proof* \rangle

end

end

4 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

Shows-Literal

begin

consts *show-real* :: *real* \Rightarrow *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*
 \langle *proof* \rangle

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

showsp-real *p r* (*x @ y*) = *showsp-real* *p r x @ y*
 \langle *proof* \rangle

\langle *ML* \rangle

derive *show real*

instantiation *real* :: *showl*

begin

definition *showsl* (*x* :: *real*) = *showsl-lit* (*String.implode* (*show-real* *x*))

```

definition showsl-list (xs :: real list) = default-showsl-list showsl xs
instance  $\langle$ proof $\rangle$ 
end

end

```

5 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 @ "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
   $\langle$ proof $\rangle$ 

```

```

lemma showsp-complex-append [show-law-simps]:
  showsp-complex p r (x @ y) = showsp-complex p r x @ y
   $\langle$ proof $\rangle$ 

```

```

 $\langle$ ML $\rangle$ 

```

```

derive show complex
end

```

6 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

We provide two parsers for natural numbers and integers, which are verified in the sense that they are the inverse of the show-function for these types. We therefore also prove that the show-functions are injective.

```

theory Number-Parser
imports
  Show-Instances
begin

```

We define here the bind-operations for option and sum-type. We do not import these operations from Certification-Monads.Strict-Sum and Parser-Monad, since these imports would yield a cyclic dependency of the two AFP entries Show and Certification-Monads.

```

definition obind where obind opt  $f = (\text{case } \text{opt} \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow f \ x)$ 
definition sbind where sbind su  $f = (\text{case } \text{su} \text{ of } \text{Inl } e \Rightarrow \text{Inl } e \mid \text{Inr } r \Rightarrow f \ r)$ 

```

context begin

A natural number parser which is proven correct:

```

definition nat-of-digit :: char  $\Rightarrow$  nat option where
  nat-of-digit  $x \equiv$ 
    if  $x = \text{CHR } "0"$  then Some 0
    else if  $x = \text{CHR } "1"$  then Some 1
    else if  $x = \text{CHR } "2"$  then Some 2
    else if  $x = \text{CHR } "3"$  then Some 3
    else if  $x = \text{CHR } "4"$  then Some 4
    else if  $x = \text{CHR } "5"$  then Some 5
    else if  $x = \text{CHR } "6"$  then Some 6
    else if  $x = \text{CHR } "7"$  then Some 7
    else if  $x = \text{CHR } "8"$  then Some 8

```

else if $x = \text{CHR } "9"$ then *Some 9*
 else *None*

private fun *nat-of-string-aux* :: *nat* \Rightarrow *string* \Rightarrow *nat option*
where

nat-of-string-aux $n \ [] = \text{Some } n \ |$
nat-of-string-aux $n \ (d \ \# \ s) = (\text{obind } (\text{nat-of-digit } d) \ (\lambda m. \ \text{nat-of-string-aux } (10 * n + m) \ s))$

definition *nat-of-string* $s \equiv$

case if $s = []$ then *None* else *nat-of-string-aux* $0 \ s$ of
None $\Rightarrow \text{Inl } (\text{STR } "cannot convert" + \text{String.implode } s + \text{STR } " to a number")$
 $| \text{Some } n \Rightarrow \text{Inr } n$

private lemma *nat-of-string-aux-snoc*:

nat-of-string-aux $n \ (s \ @ \ [c]) =$
 $\text{obind } (\text{nat-of-string-aux } n \ s) \ (\lambda l. \ \text{obind } (\text{nat-of-digit } c) \ (\lambda m. \ \text{Some } (10 * l + m)))$

<proof> **lemma** *nat-of-string-aux-digit*:

assumes $m10: m < 10$

shows *nat-of-string-aux* $n \ (s \ @ \ \text{string-of-digit } m) =$

$\text{obind } (\text{nat-of-string-aux } n \ s) \ (\lambda l. \ \text{Some } (10 * l + m))$

<proof> **lemmas** *shows-move* = *showsp-nat-append*[*of 0 - [],simplified, folded shows-prec-nat-def*]

private lemma *nat-of-string-aux-show*: *nat-of-string-aux* $0 \ (\text{show } m) = \text{Some } m$
<proof>

lemma fixes $m :: \text{nat}$ **shows** *show-nonemp*: *show* $m \neq []$
<proof>

The parser *nat-of-string* is the inverse of *show*.

lemma *nat-of-string-show[simp]*: *nat-of-string* (*show* m) = *Inr m*
<proof>

end

We also provide a verified parser for integers.

fun *safe-head* **where** *safe-head* $[] = \text{None} \ | \ \text{safe-head } (x\#xs) = \text{Some } x$

definition *int-of-string* :: *string* \Rightarrow *String.literal* + *int*

where *int-of-string* $s \equiv$

if *safe-head* $s = \text{Some } (\text{CHR } "-")$ then *sbind* (*nat-of-string* (*tl s*)) ($\lambda n. \ \text{Inr } (-\text{int } n)$)

else *sbind* (*nat-of-string* s) ($\lambda n. \ \text{Inr } (\text{int } n)$)

definition *digits* :: *char set* **where**

digits = *set* ("0123456789")

lemma *set-string-of-digit*: *set* (*string-of-digit* x) \subseteq *digits*

<proof>

lemma *range-showsp-nat*: $set (showsp\text{-}nat\ p\ n\ s) \subseteq digits \cup set\ s$
<proof>

lemma *set-show-nat*: $set (show\ (n :: nat)) \subseteq digits$
<proof>

lemma *int-of-string-show[simp]*: $int\text{-of}\text{-}string\ (show\ x) = Inr\ x$
<proof>

hide-const (**open**) *obind sbind*

Eventually, we derive injectivity of the show-functions for nat and int.

lemma *inj-show-nat*: $inj\ (show :: nat \Rightarrow string)$
<proof>

lemma *inj-show-int*: $inj\ (show :: int \Rightarrow string)$
<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/>.