Haskell's Show-Class in Isabelle/HOL*

Christian Sternagel   René Thiemann

March 19, 2025

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

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*
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 ⋀*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*
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* ⇒ *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*
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 ''[]'' ''['' '', '' '']'' xs

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*]:
$(\bigwedge x.\ x \in set\ xs \Longrightarrow show\text{-}law\ s\ x) \Longrightarrow show\text{-}law\ (showsp\text{-}list\ s)\ xs$
by (*simp add*: *show-law-def showsp-list-code show-law-simps*)

lemma *showsp-list-append* [*show-law-simps*]:
$(\bigwedge p\ y\ z.\ \forall x \in set\ xs.\ s\ p\ x\ (y\ @\ z) = s\ p\ x\ y\ @\ z) \Longrightarrow$
$showsp\text{-}list\ s\ p\ xs\ (y\ @\ z) = showsp\text{-}list\ s\ p\ xs\ y\ @\ z$
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* ''$\boxed{\hookleftarrow}$'')
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*]:
$(\bigwedge x\ y.\ s\ (x\ @\ y) = s\ x\ @\ y) \Longrightarrow shows\text{-}paren\ s\ (x\ @\ y) = shows\text{-}paren\ s\ x\ @$
y **and**
shows-quote-append [*show-law-simps*]:

$(\bigwedge x\ y.\ s\ (x\ @\ y) = s\ x\ @\ y) \Longrightarrow$ *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*:
$(\bigwedge x\ y.\ f\ (x\ @\ y) = f\ x\ @\ y) \Longrightarrow g\ (x\ @\ y) = g\ x\ @\ y \Longrightarrow (f\ o\ g)\ (x\ @\ y) = (f\ o\ g)\ x\ @\ y$
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 $\bigwedge r\ s.\ \forall x \in set\ xs.\ showx\ x\ (r\ @\ s) = showx\ x\ r\ @\ s$
shows *foldr showx xs* (*r* @ *s*) = *foldr showx xs r* @ *s*
using *assms* **by** (*induct xs*) (*simp-all*)

lemma *shows-sep-cong* [*fundef-cong*]:
assumes *xs* = *ys* **and** $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
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*

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

lemma *showsp-list-cong* [*fundef-cong*]:
$xs = ys \Longrightarrow p = q \Longrightarrow$
$(\bigwedge p\ x.\ x \in set\ ys \Longrightarrow f\ p\ x = g\ p\ x) \Longrightarrow$ *showsp-list f p xs* = *showsp-list g q ys*
by (*simp add*: *showsp-list-code cong*: *shows-list-gen-cong*)

abbreviation (*input*) *shows-cons* :: *string* ⇒ *shows* ⇒ *shows* (**infixr** ‹+#+› *10*)
where
s +#+ *p* ≡ *shows-string s* ∘ *p*

abbreviation (*input*) *shows-append* :: *shows* ⇒ *shows* ⇒ *shows* (**infixr** ‹+@+›
10)
where
s +@+ *p* ≡ *s* ∘ *p*

instantiation *String.literal* :: *show*
begin

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

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

lemma *shows-list-literal-code* [*code*]:
shows-list = *foldr* (*λs. shows-string* (*String.explode s*))
proof
fix *ss*
show *shows-list ss* = *foldr* (*λs. 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.

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 ≡ *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 *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* '')''
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

definition *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*: *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 ‹
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*} @{*thm show-law-bool*}
#> *Show-Generator.register-foreign-showsp* @{*typ char*} @{*term showsp-char*} @{*thm show-law-char*}
#> *Show-Generator.register-foreign-showsp* @{*typ nat*} @{*term showsp-nat*} @{*thm show-law-nat*}
#> *Show-Generator.register-foreign-showsp* @{*typ int*} @{*term showsp-int*} @{*thm show-law-int*}
#> *Show-Generator.register-foreign-showsp* @{*typ rat*} @{*term showsp-rat*} @{*thm show-law-rat*}
›

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*
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 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* ⇒ *String.literal*

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

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

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

fun *showsl-sep* :: ('*a* ⇒ *showsl*) ⇒ *showsl* ⇒ '*a list* ⇒ *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* ⇒ *showsl*) ⇒ *String.literal* ⇒ *String.literal*
⇒ *String.literal* ⇒ *String.literal* ⇒ '*a list* ⇒ *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* ⇒ *showsl*) ⇒ '*a list* ⇒ *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*'') **by** *code-simp*

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

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

definition *showsl-lines desc-empty* = *showsl-list-gen showsl desc-empty* (*STR* '''') (*STR* ''$\boxed{\hookleftarrow}$'') (*STR* '''')

abbreviation *showl* **where** *showl x* ≡ *showsl x* (*STR* '''')

instantiation *char* :: *showl*
begin
definition *showsl-char c* = *showsl-lit* (*String.implode* [*c*]) — Shouldn't there be a faster conversion than via strings?
definition *showsl-list-char cs s* = *showsl-lit* (*String.implode cs*) *s*
instance ..

end

instantiation *String.literal* :: *showl*
begin
definition *showsl* (*s* :: *String.literal*) = *showsl-lit s*
definition *showsl-list* (*xs* :: *String.literal list*) = *default-showsl-list showsl xs*
instance ..
end

instantiation *bool* :: *showl*
begin
definition *showsl* (*b* :: *bool*) = *showsl-lit* (*if b then STR ''True'' else STR ''False''*)

definition *showsl-list* (*xs* :: *bool list*) = *default-showsl-list showsl xs*
instance ..
end

instantiation *nat* :: *showl*
begin
fun *showsl-nat* :: *nat* ⇒ *showsl* **where**
  showsl-nat n =
    (*if n* < *10 then showsl-lit* (*lit-of-digit n*)
    else showsl-nat (*n div 10*) *o showsl-lit* (*lit-of-digit* (*n mod 10*)))
definition *showsl-list* (*xs* :: *nat list*) = *default-showsl-list showsl xs*
instance ..
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* :: *int list*) = *default-showsl-list showsl xs*
instance ..
end

instantiation *integer* :: *showl*
begin
definition *showsl-integer* :: *integer* ⇒ *showsl* **where** *showsl-integer i* = *showsl* (*int-of-integer i*)
definition *showsl-list-integer* :: *integer list* ⇒ *showsl* **where** *showsl-list-integer xs* = *default-showsl-list showsl xs*
instance ..
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 ..
end

instantiation *unit :: showl*
begin
definition *showsl (x :: unit) = showsl-lit (STR ''()'')*
definition *showsl-list (xs :: unit list) = default-showsl-list showsl xs*
instance ..
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 ..
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 ..
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 ..
end

definition *[code-unfold]: showsl-nl = showsl (STR ''$\boxed{\hookleftarrow}$'')*

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

instantiation *list :: (showl) showl*

begin
definition *showsl-list* :: *'a list* ⇒ *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 ..
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* ⇒ *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
  by (*rule show-lawI*) (*simp add*: *showsp-real-def show-law-simps*)

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

local-setup ‹
  Show-Generator.register-foreign-showsp @{*typ real*} @{*term showsp-real*} @{*thm show-law-real*}
›

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 ..

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
  by (*rule show-lawI*) (*simp add*: *showsp-complex-def show-law-simps*)

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

local-setup ‹
 Show-Generator.register-foreign-showsp @{*typ complex*} @{*term showsp-complex*} @{*thm show-law-complex*}
›

derive *show complex*
end

6 Show Implemetation 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

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* = (*case opt of None* ⇒ *None* | *Some x* ⇒ *f x*)
definition *sbind* **where** *sbind su f* = (*case su of Inl e* ⇒ *Inl e* | *Inr r* ⇒ *f r*)

context begin

A natural number parser which is proven correct:

definition *nat-of-digit* :: *char* ⇒ *nat option* **where**
  nat-of-digit x ≡
    if x = *CHR ''0'' then Some 0*
    else if x = *CHR ''1'' then Some 1*
    else if x = *CHR ''2'' then Some 2*

else if x = CHR ''3'' then Some 3
else if x = CHR ''4'' then Some 4
else if x = CHR ''5'' then Some 5
else if x = CHR ''6'' then Some 6
else if x = CHR ''7'' then Some 7
else if x = CHR ''8'' then Some 8
else if x = CHR ''9'' then Some 9
else None

private fun *nat-of-string-aux* :: *nat* ⇒ *string* ⇒ *nat option*
where
nat-of-string-aux n [] = *Some n* |
nat-of-string-aux n (*d* # *s*) = (*obind* (*nat-of-digit d*) (λ*m*. *nat-of-string-aux* (*10* ∗ *n* + *m*) *s*))

definition *nat-of-string s* ≡
case if s = [] *then None else nat-of-string-aux 0 s of*
None ⇒ *Inl* (*STR ''cannot convert ''* + *String.implode s* + *STR '' to a number''*)
| *Some n* ⇒ *Inr n*

private lemma *nat-of-string-aux-snoc*:
nat-of-string-aux n (*s* @ [*c*]) =
obind (*nat-of-string-aux n s*) (λ *l*. *obind* (*nat-of-digit c*) (λ *m*. *Some* (*10* ∗ *l* + *m*)))
by (*induct s arbitrary*:*n*, *auto simp*: *obind-def split*: *option.splits*)

private lemma *nat-of-string-aux-digit*:
assumes *m10*: *m* < *10*
shows *nat-of-string-aux n* (*s* @ *string-of-digit m*) =
obind (*nat-of-string-aux n s*) (λ *l*. *Some* (*10* ∗ *l* + *m*))
proof −
from *m10* **have** *m* = *0* ∨ *m* = *1* ∨ *m* = *2* ∨ *m* = *3* ∨ *m* = *4* ∨ *m* = *5* ∨ *m* = *6* ∨ *m* = *7* ∨ *m* = *8* ∨ *m* = *9*
by *presburger*
thus *?thesis* **by** (*auto simp add*: *nat-of-digit-def nat-of-string-aux-snoc string-of-digit-def obind-def split*: *option.splits*)
qed

private 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* (*show m*) = *Some m*
proof (*induct m rule*:*less-induct*)
case *IH*: (*less m*)
show *?case* **proof** (*cases m* < *10*)
case *m10*: *True*
show *?thesis*
apply (*unfold shows-prec-nat-def*)
apply (*subst showsp-nat.simps*)

using *m10 nat-of-string-aux-digit*[*OF m10*, *of 0* []]
by (*auto simp add*:*shows-string-def nat-of-string-def string-of-digit-def obind-def*)
next
case *m*: *False*
then have *m div 10* < *m* **by** *auto*
note *IH* = *IH*[*OF this*]
show *?thesis* **apply** (*unfold shows-prec-nat-def*, *subst showsp-nat.simps*)
using *m* **apply** (*simp add*: *shows-prec-nat-def*[*symmetric*] *shows-string-def*)
apply (*subst shows-move*)
using *nat-of-string-aux-digit m IH*
by (*auto simp*: *nat-of-string-def obind-def*)
qed
qed

lemma fixes *m* :: *nat* **shows** *show-nonemp*: *show m* ≠ []
apply (*unfold shows-prec-nat-def*)
apply (*subst showsp-nat.simps*)
apply (*fold shows-prec-nat-def*)
apply (*unfold o-def*)
apply (*subst shows-move*)
apply (*auto simp*: *shows-string-def string-of-digit-def*)
done

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

lemma *nat-of-string-show*[*simp*]: *nat-of-string* (*show m*) = *Inr m*
using *nat-of-string-aux-show* **by** (*auto simp*: *nat-of-string-def show-nonemp*)

end

We also provide a verified parser for integers.

fun *safe-head* **where** *safe-head* [] = *None* | *safe-head* (*x#xs*) = *Some x*

definition *int-of-string* :: *string* ⇒ *String.literal* + *int*
where *int-of-string s* ≡
if safe-head s = *Some* (*CHR ''−''*) *then sbind* (*nat-of-string* (*tl s*)) (λ *n*. *Inr* (− *int n*))
else sbind (*nat-of-string s*) (λ *n*. *Inr* (*int n*))

definition *digits* :: *char set* **where**
digits = *set* (*''0123456789''*)

lemma *set-string-of-digit*: *set* (*string-of-digit x*) ⊆ *digits*
unfolding *digits-def string-of-digit-def* **by** *auto*

lemma *range-showsp-nat*: *set* (*showsp-nat p n s*) ⊆ *digits* ∪ *set s*
proof (*induct p n arbitrary*: *s rule*: *showsp-nat.induct*)
case (*1 p n s*)
then show *?case* **using** *set-string-of-digit*[*of n*] *set-string-of-digit*[*of n mod 10*]
by (*auto simp*: *showsp-nat.simps*[*of p n*] *shows-string-def*) *fastforce*

qed

lemma *set-show-nat*: *set* (*show* (*n* :: *nat*)) $\subseteq$ *digits*
  using *range-showsp-nat*[*of 0 n Nil*] **unfolding** *shows-prec-nat-def* **by** *auto*

lemma *int-of-string-show*[*simp*]: *int-of-string* (*show x*) = *Inr x*
proof −
  have *show x* = *showsp-int 0 x* []
    by (*simp add*: *shows-prec-int-def*)
  also have ... = (*if x* < *0 then* ''−'' @ *show* (*nat* (−*x*)) *else show* (*nat x*))
    unfolding *showsp-int-def if-distrib shows-prec-nat-def*
    by (*simp add*: *shows-string-def*)
  also have *int-of-string* ... = *Inr x*
  proof (*cases x* < *0*)
    case *True*
    thus *?thesis* **unfolding** *int-of-string-def sbind-def* **by** *simp*
  next
    case *False*
    from *set-show-nat* **have** *set* (*show* (*nat x*)) $\subseteq$ *digits* **.**
    hence *CHR* ''−'' $\notin$ *set* (*show* (*nat x*)) **unfolding** *digits-def* **by** *auto*
    hence *safe-head* (*show* (*nat x*)) $\neq$ *Some CHR* ''−''
      by (*cases show* (*nat x*), *auto*)
    thus *?thesis* **using** *False*
      by (*simp add*: *int-of-string-def sbind-def*)
  qed
  finally show *?thesis* **.**
qed

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*)
  by (*rule inj-on-inverseI*[*of - λ s. case nat-of-string s of Inr x* $\Rightarrow$ *x*], *auto*)

lemma *inj-show-int*: *inj* (*show* :: *int* $\Rightarrow$ *string*)
  by (*rule inj-on-inverseI*[*of - λ s. case int-of-string s of Inr x* $\Rightarrow$ *x*], *auto*)

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/.