Contents

1 Bool

```
(*****************************************************************************
(* Boolean                                                                   *)
(*****************************************************************************)

(* rename module to clash with existing list modules of targets *)

declare {isabelle, hol, ocaml, coq} rename module = lem_bool

(* The type bool is hard-coded, so are true and false *)

(* ------------------------ *)
(* not                      *)
(* ------------------------ *)

val not : BOOL → BOOL
let not b = match b with
  | true → false
  | false → true
end

declare hol target_rep function not x = '~' x
declare ocaml target_rep function not = 'not'
declare isabelle target_rep function not x = '\<not>' x
declare html target_rep function not = '&not;'
declare coq target_rep function not = 'negb'
declare tex target_rep function not b = '$\neg$' b

assert not1 : ¬ (¬ true)
assert not2 : ¬ false

(* ------------------------ *)
(* and                      *)
(* ------------------------ *)

val && [and] : BOOL → BOOL → BOOL
let && b1 b2 = match (b1, b2) with
  | (true, true) → true
  | _ → false
end

declare hol target_rep function and = infix right_assoc 0 '/\'
declare ocaml target_rep function and = infix '&&'
declare isabelle target_rep function and = infix '\<and>'
declare coq target_rep function and = infix '&&'
declare html target_rep function and = infix '&and;'
declare tex target_rep function and = infix '$\wedge$'

assert and1 : (¬ (true ∧ false))
assert and2 : (¬ (false ∧ true))
assert and3 : (¬ (false ∧ false))
assert and4 : (true ∧ true)


(* ------------------------ *)
(* or                       *)
(* ------------------------ *)
```

```
val || [or] : BOOL → BOOL → BOOL
let || b1 b2 = match (b1, b2) with
  | (false, false) → false
  | _ → true
end

declare hol target_rep function or = infix '\/'
declare ocaml target_rep function or = infix '||'
declare isabelle target_rep function or = infix '\<or>'
declare coq target_rep function or = infix '||'
declare html target_rep function or = infix '&or;'
declare tex target_rep function or = infix '$\vee$'

assert or1 : (true ∨ false)
assert or2 : (false ∨ true)
assert or3 : (true ∨ true)
assert or4 : (¬ (false ∨ false))


(* ---------------------------------- *)
(* implication          *)
(* ---------------------------------- *)

val --> [imp] : BOOL → BOOL → BOOL
let --> b1 b2 = match (b1, b2) with
  | (true, false) → false
  | _ → true
end

declare hol target_rep function imp = infix '==>'
declare isabelle target_rep function imp = infix '\<longrightarrow>'
(* declare coq     target_rep function (--> ) = `imp` *)
declare html target_rep function imp = infix '&rarr;'
declare tex target_rep function imp = infix '$\longrightarrow$'

let inline {ocaml, coq} imp x y = ((¬ x) ∨ y)

assert imp1 : (¬ (true ⟶ false))
assert imp2 : (false ⟶ true)
assert imp3 : (false ⟶ false)
assert imp4 : (true ⟶ true)


(* ---------------------------------- *)
(* equivalence          *)
(* ---------------------------------- *)

val <--> [equiv] : BOOL → BOOL → BOOL
let <--> b1 b2 = match (b1, b2) with
  | (true, true) → true
  | (false, false) → true
  | _ → false
end

declare hol target_rep function equiv = infix '<=>'
declare isabelle target_rep function equiv = infix '\<longleftrightarrow>'
declare coq target_rep function equiv = 'Bool.eqb'
```

declare $ocaml$ target_rep function $\mathrm{equiv}\ =$ infix '='
declare $html$ target_rep function $\mathrm{equiv}\ =$ infix '↔'
declare tex target_rep function $\mathrm{equiv}\ =$ infix '$\longleftrightarrow$'

assert $equiv_1$: $(\neg\ (\mathsf{true} \longleftrightarrow \mathsf{false}))$
assert $equiv_2$: $(\neg\ (\mathsf{false} \longleftrightarrow \mathsf{true}))$
assert $equiv_3$: $(\mathsf{false} \longleftrightarrow \mathsf{false})$
assert $equiv_4$: $(\mathsf{true} \longleftrightarrow \mathsf{true})$

(* - *)
(* xor *)
(* - *)

val xor : BOOL $\rightarrow$ BOOL $\rightarrow$ BOOL
let inline $xor\ b_1\ b_2\ =\ \neg\ (b_1 \longleftrightarrow b_2)$

assert xor_1 : $(\mathrm{xor}\ \mathsf{true}\ \mathsf{false})$
assert xor_2 : $(\mathrm{xor}\ \mathsf{false}\ \mathsf{true})$
assert xor_3 : $(\neg\ (\mathrm{xor}\ \mathsf{true}\ \mathsf{true}))$
assert xor_4 : $(\neg\ (\mathrm{xor}\ \mathsf{false}\ \mathsf{false}))$

2 Basic_classes

```
(* ************************************************************************ *)
(* Basic Type Classes                                                       *)
(* ************************************************************************ *)

open import Bool

declare {isabelle, ocaml, hol, coq} rename module = lem_basic_classes

open import {coq} Coq.Strings.Ascii
open import {hol} ternaryComparisonsTheory

(* ========================================================================== *)
(* Equality                                                                   *)
(* ========================================================================== *)

(* Lem's default equality (=) is defined by the following type-class Eq.  This typeclass should define equal

class ( Eq α )
  val = [isEqual] : α → α → BOOL
  val <> [isInequal] : α → α → BOOL
end

declare coq target_rep function isEqual = infix '='
(* declare coq target_rep function isEqual = infix `=`declare coq target_rep function isInequal = infix `<>` *)
declare tex target_rep function isInequal = infix '$\neq$'


(* (=) should for all instances be an equivalence relation  The isEquivalence predicate of relations could b

(* TODO: add later, once lemmata can be assigned to classeslemma eq_equiv: ((forall x. (x = x)) &&

(* Structural equality *)

(* Sometimes, it is also handy to be able to use structural equality.  This equality is mapped to the build-
val unsafe_structural_equality : ∀ α. α → α → BOOL

declare hol target_rep function unsafe_structural_equality = infix '='
declare ocaml target_rep function unsafe_structural_equality = infix '='
declare isabelle target_rep function unsafe_structural_equality = infix '='
declare coq target_rep function unsafe_structural_equality = 'classical_boolean_equivalence'

val unsafe_structural_inequality : ∀ α. α → α → BOOL
let unsafe_structural_inequality x y = ¬ (unsafe_structural_equality x y)
declare isabelle target_rep function unsafe_structural_inequality = infix '\<noteq>'
declare hol target_rep function unsafe_structural_inequality = infix '<>'

(* The default for equality is the unsafe structural one. It can  (and should) be overriden for concrete types

default_instance ∀ α. (Eq α)
  let = = unsafe_structural_equality
  let <> = unsafe_structural_inequality
end

(* for HOL and Isabelle, be even stronger and always(!) use  standard equality *)
let inline {hol, isabelle} = = unsafe_structural_equality
let inline {hol, isabelle} <> = unsafe_structural_inequality
```

```
(* ========================================================================== *)
(* Orderings                                                                  *)
(* ========================================================================== *)

(* The type - class Ord represents total orders (also called linear orders) *)
type ORDERING = LT | EQ | GT

declare ocaml target_rep type ORDERING = 'int'
declare ocaml target_rep function LT = '(-1)'
declare ocaml target_rep function EQ = '0'
declare ocaml target_rep function GT = '1'

declare coq target_rep type ORDERING = 'ordering'
declare coq target_rep function LT = 'LT'
declare coq target_rep function EQ = 'EQ'
declare coq target_rep function GT = 'GT'

declare hol target_rep type ORDERING = 'ordering'
declare hol target_rep function LT = 'LESS'
declare hol target_rep function EQ = 'EQUAL'
declare hol target_rep function GT = 'GREATER'

let orderingIsLess r = (match r with LT -> true | _ -> false end)
let orderingIsGreater r = (match r with GT -> true | _ -> false end)
let orderingIsEqual r = (match r with EQ -> true | _ -> false end)
let inline orderingIsLessEqual r = ¬ (orderingIsGreater r)
let inline orderingIsGreaterEqual r = ¬ (orderingIsLess r)

let ordering_cases r lt eq gt =
  if orderingIsLess r then lt else
  if orderingIsEqual r then eq else gt

declare ocaml target_rep function orderingIsLess = 'Lem.orderingIsLess'
declare ocaml target_rep function orderingIsGreater = 'Lem.orderingIsGreater'
declare ocaml target_rep function orderingIsEqual = 'Lem.orderingIsEqual'

declare ocaml target_rep function ordering_cases = 'Lem.ordering_cases'

declare {ocaml} pattern_match exhaustive ORDERING = [ LT; EQ ; GT ] ordering_cases

assert ordering_cases_0 : (ordering_cases LT true false false)
assert ordering_cases_1 : (ordering_cases EQ false true false)
assert ordering_cases_2 : (ordering_cases GT false false true)
assert ordering_match_1 : (match LT with GT -> false ∧ false | _ -> true end)
assert ordering_match_2 : (match EQ with GT -> false | _ -> true end)
assert ordering_match_3 : (match GT with GT -> true ∧ true | _ -> false end)
assert ordering_match_4 : ((fun r -> (match r with GT -> false | _ -> true end)) LT)
assert ordering_match_5 : ((fun r -> (match r with GT -> false | _ -> true end)) EQ)
assert ordering_match_6 : ((fun r -> (match r with GT -> true ∧ true | _ -> false end)) GT)


val orderingEqual : ORDERING -> ORDERING -> BOOL
let inline ~{ocaml, coq} orderingEqual = unsafe_structural_equality
declare coq target_rep function orderingEqual left right = ('ordering_equal' left right)
declare ocaml target_rep function orderingEqual = 'Lem.orderingEqual'
```

```
instance (Eq ORDERING)
  let = =  orderingEqual
  let <> x y =  ¬ (orderingEqual x y)
end

class ( Ord α )
  val compare : α → α → ORDERING
  val < [isLess] : α → α → BOOL
  val <= [isLessEqual] : α → α → BOOL
  val > [isGreater] : α → α → BOOL
  val >= [isGreaterEqual] : α → α → BOOL
end

declare coq target_rep function isLess = 'isLess'
declare coq target_rep function isLessEqual = 'isLessEqual'
declare coq target_rep function isGreater = 'isGreater'
declare coq target_rep function isGreaterEqual = 'isGreaterEqual'
declare tex target_rep function isLess = infix '$<$'
declare tex target_rep function isLessEqual = infix '$\le$'
declare tex target_rep function isGreater = infix '$>$'
declare tex target_rep function isGreaterEqual = infix '$\ge$'


(* Ocaml provides default, polymorphic compare functions. Let's use them  as the default. However, because us
val defaultCompare : ∀ α. α → α → ORDERING
val defaultLess : ∀ α. α → α → BOOL
val defaultLessEq : ∀ α. α → α → BOOL
val defaultGreater : ∀ α. α → α → BOOL
val defaultGreaterEq : ∀ α. α → α → BOOL

declare ocaml target_rep function defaultCompare = 'compare'
declare hol target_rep function defaultCompare =
declare isabelle target_rep function defaultCompare =
declare coq target_rep function defaultCompare x y = EQ

declare ocaml target_rep function defaultLess = infix '<'
declare hol target_rep function defaultLess =
declare isabelle target_rep function defaultLess =
declare coq target_rep function defaultLess =

declare ocaml target_rep function defaultLessEq = infix '<='
declare hol target_rep function defaultLessEq =
declare isabelle target_rep function defaultLessEq =
declare coq target_rep function defaultLessEq =

declare ocaml target_rep function defaultGreater = infix '>'
declare hol target_rep function defaultGreater =
declare isabelle target_rep function defaultGreater =
declare coq target_rep function defaultGreater =

declare ocaml target_rep function defaultGreaterEq = infix '>='
declare hol target_rep function defaultGreaterEq =
declare isabelle target_rep function defaultGreaterEq =
declare coq target_rep function defaultGreaterEq =


let genericCompare (less : α → α → BOOL) (equal : α → α → BOOL) (x : α) (y : α) =
  if less x y then
```

```
    LT
  else if equal x y then
    EQ
  else
    GT


(*(* compare should really be a total order *)lemma ord_OK_1 : (  (forall x y. (compare x y = EQ) <-> (compare y

(* let's derive a compare function from the Ord type - class *)
val ordCompare : ∀ α. Eq α, Ord α ⇒ α → α → ORDERING
let ordCompare x y =
   if (x < y) then LT else
  if (x = y) then EQ else GT

class ( OrdMaxMin α )
  val max : α → α → α
  val min : α → α → α
end

val minByLessEqual : ∀ α. (α → α → BOOL) → α → α → α
let ~{isabelle} minByLessEqual le x y =  if (le x y) then x else y
let inline {isabelle} minByLessEqual le x y  =  if (le x y) then x else y

val maxByLessEqual : ∀ α. (α → α → BOOL) → α → α → α
let ~{isabelle} maxByLessEqual le x y =  if (le y x) then x else y
let inline {isabelle} maxByLessEqual le x y  =  if (le y x) then x else y

val defaultMax : ∀ α. Ord α ⇒ α → α → α
let inline defaultMax  =  maxByLessEqual (≤)
declare ocaml target_rep function defaultMax  =  'max'

val defaultMin : ∀ α. Ord α ⇒ α → α → α
let inline defaultMin  =  minByLessEqual (≤)
declare ocaml target_rep function defaultMin  =  'min'

default_instance ∀ α. Ord α ⇒ ( OrdMaxMin α)
  let max =  defaultMax
  let min =  defaultMin
end


(* ========================================================================== *)
(* SetTypes                                                                   *)
(* ========================================================================== *)

(* Set implementations use often an order on the elements. This allows the OCaml implementation  to use trees

class ( SetType α )
  val {ocaml, coq} setElemCompare : α → α → ORDERING
end

default_instance ∀ α. ( SetType α )
  let setElemCompare =  defaultCompare
end

(* ========================================================================== *)
(* Instantiations                                                             *)
```

```
(* ========================================================================== *)

instance (Eq BOOL)
  let = =  (⟷)
  let <> x y =  ¬ ((⟷) x y)
end

let boolCompare b1 b2 =  match (b1, b2) with
  | (true, true) → EQ
  | (true, false) → GT
  | (false, true) → LT
  | (false, false) → EQ
end

instance (SetType BOOL)
  let setElemCompare =  boolCompare
end

(* strings *)

val charEqual : CHAR → CHAR → BOOL
let inline ~{coq} charEqual = unsafe_structural_equality
declare coq target_rep function charEqual left right = ('char_equal' left right)

instance (Eq CHAR)
  let = =  charEqual
  let <> left right =  ¬ (charEqual left right)
end

val stringEquality : STRING → STRING → BOOL
declare coq target_rep function stringEquality left right = ('string_equal' left right)
let inline {ocaml, hol, isabelle} stringEquality = unsafe_structural_equality

instance (Eq STRING)
  let = =  stringEquality
  let <> l r =  ¬ (stringEquality l r)
end

(* pairs *)

val pairEqual : ∀ α β. Eq α, Eq β ⇒ (α * β) → (α * β) → BOOL
let pairEqual (a1, b1) (a2, b2) =  (a1 = a2) ∧ (b1 = b2)

val pairEqualBy : ∀ α β. (α → α → BOOL) → (β → β → BOOL) → (α * β) → (α * β) → BOOL
declare ocaml target_rep function pairEqualBy = 'Lem.pair_equal'
declare coq target_rep function pairEqualBy leftEq rightEq left right = ('tuple_equal_by' leftEq rightEq left right)


let inline {hol, isabelle} pairEqual = unsafe_structural_equality
let inline {ocaml, coq} pairEqual = pairEqualBy (=) (=)

instance ∀ α β. Eq α, Eq β ⇒ (Eq (α * β))
  let = =  pairEqual
  let <> x y =  ¬ (pairEqual x y)
end

val pairCompare : ∀ α β. (α → α → ORDERING) → (β → β → ORDERING) → (α * β) →
(α * β) → ORDERING
```

```
let pairCompare cmpa cmpb (a1, b1) (a2, b2) =
  match cmpa a1 a2 with
  | LT → LT
  | GT → GT
  | EQ → cmpb b1 b2
  end

let pairLess (x1, x2) (y1, y2) = (x1 < y1) ∨ ((x1 ≤ y1) ∧ (x2 < y2))
let pairLessEq (x1, x2) (y1, y2) = (x1 < y1) ∨ ((x1 ≤ y1) ∧ (x2 ≤ y2))

let pairGreater x12 y12 = pairLess y12 x12
let pairGreaterEq x12 y12 = pairLessEq y12 x12

instance ∀ α β. Ord α, Ord β ⇒ (Ord (α * β))
  let compare = pairCompare compare compare
  let < = pairLess
  let <= = pairLessEq
  let > = pairGreater
  let >= = pairGreaterEq
end

instance ∀ α β. SetType α, SetType β ⇒ (SetType (α * β))
  let setElemCompare = pairCompare setElemCompare setElemCompare
end


(* triples *)

val tripleEqual : ∀ α β γ. Eq α, Eq β, Eq γ ⇒ (α * β * γ) → (α * β * γ) → BOOL
let tripleEqual (x1, x2, x3) (y1, y2, y3) = ((x1, (x2, x3)) = (y1, (y2, y3)))
let inline {hol, isabelle} tripleEqual = unsafe_structural_equality

instance ∀ α β γ. Eq α, Eq β, Eq γ ⇒ (Eq (α * β * γ))
  let = = tripleEqual
  let <> x y = ¬ (tripleEqual x y)
end

val tripleCompare : ∀ α β γ. (α → α → ORDERING) → (β → β → ORDERING) → (γ → γ →
ORDERING) → (α * β * γ) → (α * β * γ) → ORDERING
let tripleCompare cmpa cmpb cmpc (a1, b1, c1) (a2, b2, c2) =
  pairCompare cmpa (pairCompare cmpb cmpc) (a1, (b1, c1)) (a2, (b2, c2))

let tripleLess (x1, x2, x3) (y1, y2, y3) = (x1, (x2, x3)) < (y1, (y2, y3))
let tripleLessEq (x1, x2, x3) (y1, y2, y3) = (x1, (x2, x3)) ≤ (y1, (y2, y3))

let tripleGreater x123 y123 = tripleLess y123 x123
let tripleGreaterEq x123 y123 = tripleLessEq y123 x123

instance ∀ α β γ. Ord α, Ord β, Ord γ ⇒ (Ord (α * β * γ))
  let compare = tripleCompare compare compare compare
  let < = tripleLess
  let <= = tripleLessEq
  let > = tripleGreater
  let >= = tripleGreaterEq
end

instance ∀ α β γ. SetType α, SetType β, SetType γ ⇒ (SetType (α * β * γ))
  let setElemCompare = tripleCompare setElemCompare setElemCompare setElemCompare
```

```
end

(* quadruples *)

val quadrupleEqual : ∀ α β γ δ. Eq α, Eq β, Eq γ, Eq δ ⇒ (α * β * γ * δ) → (α * β * γ * δ) → BOOL

let quadrupleEqual (x1, x2, x3, x4) (y1, y2, y3, y4) = ((x1, (x2, (x3, x4))) = (y1, (y2, (y3, y4))))
let inline {hol, isabelle} quadrupleEqual = unsafe_structural_equality

instance ∀ α β γ δ. Eq α, Eq β, Eq γ, Eq δ ⇒ (Eq (α * β * γ * δ))
  let = = quadrupleEqual
  let <> x y = ¬ (quadrupleEqual x y)
end

val quadrupleCompare : ∀ α β γ δ. (α → α → ORDERING) → (β → β → ORDERING) → (γ →
γ → ORDERING) →
                                (δ → δ → ORDERING) → (α * β * γ * δ) → (α * β * γ * δ) → ORDERING

let quadrupleCompare cmpa cmpb cmpc cmpd (a1, b1, c1, d1) (a2, b2, c2, d2) =
  pairCompare cmpa (pairCompare cmpb (pairCompare cmpc cmpd)) (a1, (b1, (c1, d1))) (a2, (b2, (c2, d2)))


let quadrupleLess (x1, x2, x3, x4) (y1, y2, y3, y4) = (x1, (x2, (x3, x4))) < (y1, (y2, (y3, y4)))
let quadrupleLessEq (x1, x2, x3, x4) (y1, y2, y3, y4) = (x1, (x2, (x3, x4))) ≤ (y1, (y2, (y3, y4)))

let quadrupleGreater x1234 y1234 = quadrupleLess y1234 x1234
let quadrupleGreaterEq x1234 y1234 = quadrupleLessEq y1234 x1234

instance ∀ α β γ δ. Ord α, Ord β, Ord γ, Ord δ ⇒ (Ord (α * β * γ * δ))
  let compare = quadrupleCompare compare compare compare compare
  let < = quadrupleLess
  let <= = quadrupleLessEq
  let > = quadrupleGreater
  let >= = quadrupleGreaterEq
end

instance ∀ α β γ δ. SetType α, SetType β, SetType γ, SetType δ ⇒ (SetType (α * β * γ * δ))
  let setElemCompare = quadrupleCompare setElemCompare setElemCompare setElemCompare setElemCompare
end

(* quintuples *)

val quintupleEqual : ∀ α β γ δ 'e. Eq α, Eq β, Eq γ, Eq δ, Eq 'e ⇒ (α * β * γ * δ * 'e) →
(α * β * γ * δ * 'e) → BOOL
let quintupleEqual (x1, x2, x3, x4, x5) (y1, y2, y3, y4, y5) = ((x1, (x2, (x3, (x4, x5)))) = (y1, (y2, (y3, (y4, y5)))))

let inline {hol, isabelle} quintupleEqual = unsafe_structural_equality

instance ∀ α β γ δ 'e. Eq α, Eq β, Eq γ, Eq δ, Eq 'e ⇒ (Eq (α * β * γ * δ * 'e))
  let = = quintupleEqual
  let <> x y = ¬ (quintupleEqual x y)
end

val quintupleCompare : ∀ α β γ δ 'e. (α → α → ORDERING) → (β → β → ORDERING) → (γ →
γ → ORDERING) →
                                (δ → δ → ORDERING) → ('e → 'e → ORDERING) →
(α * β * γ * δ * 'e) → (α * β * γ * δ * 'e) → ORDERING
let quintupleCompare cmpa cmpb cmpc cmpd cmpe (a1, b1, c1, d1, e1) (a2, b2, c2, d2, e2) =
```

```
  pairCompare cmpa (pairCompare cmpb (pairCompare cmpc (pairCompare cmpd cmpe))) (a_1, (b_1, (c_1, (d_1, e_1)))) (a_2, (b

let quintupleLess (x_1, x_2, x_3, x_4, x_5) (y_1, y_2, y_3, y_4, y_5) = (x_1, (x_2, (x_3, (x_4, x_5)))) < (y_1, (y_2, (y_3, (y_4, y_5))))

let quintupleLessEq (x_1, x_2, x_3, x_4, x_5) (y_1, y_2, y_3, y_4, y_5) = (x_1, (x_2, (x_3, (x_4, x_5)))) ≤ (y_1, (y_2, (y_3, (y_4, y_5))))

let quintupleGreater x_12345 y_12345 = quintupleLess y_12345 x_12345
let quintupleGreaterEq x_12345 y_12345 = quintupleLessEq y_12345 x_12345

instance ∀ α β γ δ 'e. Ord α, Ord β, Ord γ, Ord δ, Ord 'e ⇒ (Ord (α * β * γ * δ * 'e))
  let compare = quintupleCompare compare compare compare compare compare
  let < = quintupleLess
  let <= = quintupleLessEq
  let > = quintupleGreater
  let >= = quintupleGreaterEq
end

instance ∀ α β γ δ 'e. SetType α, SetType β, SetType γ, SetType δ, SetType 'e ⇒ (SetType (α * β * γ * δ * 'e))
  let setElemCompare = quintupleCompare setElemCompare setElemCompare setElemCompare setElemCompare setElemC
end

(* sextuples *)

val sextupleEqual : ∀ α β γ δ 'e 'f. Eq α, Eq β, Eq γ, Eq δ, Eq 'e, Eq 'f ⇒ (α * β * γ * δ * 'e * 'f) →
(α * β * γ * δ * 'e * 'f) → BOOL
let sextupleEqual (x_1, x_2, x_3, x_4, x_5, x_6) (y_1, y_2, y_3, y_4, y_5, y_6) = ((x_1, (x_2, (x_3, (x_4, (x_5, x_6)))))) = (y_1, (y_2, (y_3, (y_4,

let inline {hol, isabelle} sextupleEqual = unsafe_structural_equality

instance ∀ α β γ δ 'e 'f. Eq α, Eq β, Eq γ, Eq δ, Eq 'e, Eq 'f ⇒ (Eq (α * β * γ * δ * 'e * 'f))
  let = = sextupleEqual
  let <> x y = ¬ (sextupleEqual x y)
end

val sextupleCompare : ∀ α β γ δ 'e 'f. (α → α → ORDERING) → (β → β → ORDERING) → (γ →
γ → ORDERING) →
                                (δ → δ → ORDERING) → ('e → 'e → ORDERING) → ('f → 'f →
ORDERING) →
                                (α * β * γ * δ * 'e * 'f) → (α * β * γ * δ * 'e * 'f) → ORDERING
let sextupleCompare cmpa cmpb cmpc cmpd cmpe cmpf (a_1, b_1, c_1, d_1, e_1, f_1) (a_2, b_2, c_2, d_2, e_2, f_2) =
  pairCompare cmpa (pairCompare cmpb (pairCompare cmpc (pairCompare cmpd (pairCompare cmpe cmpf)))) (a_1, (b_1, (c

let sextupleLess (x_1, x_2, x_3, x_4, x_5, x_6) (y_1, y_2, y_3, y_4, y_5, y_6) = (x_1, (x_2, (x_3, (x_4, (x_5, x_6))))) < (y_1, (y_2, (y_3, (y_4,

let sextupleLessEq (x_1, x_2, x_3, x_4, x_5, x_6) (y_1, y_2, y_3, y_4, y_5, y_6) = (x_1, (x_2, (x_3, (x_4, (x_5, x_6))))) ≤ (y_1, (y_2, (y_3, (y_4

let sextupleGreater x_123456 y_123456 = sextupleLess y_123456 x_123456
let sextupleGreaterEq x_123456 y_123456 = sextupleLessEq y_123456 x_123456

instance ∀ α β γ δ 'e 'f. Ord α, Ord β, Ord γ, Ord δ, Ord 'e, Ord 'f ⇒ (Ord (α * β * γ * δ * 'e * 'f))
  let compare = sextupleCompare compare compare compare compare compare compare
  let < = sextupleLess
  let <= = sextupleLessEq
  let > = sextupleGreater
```

```
  let >= =  sextupleGreaterEq
end

instance ∀ α β γ δ 'e 'f. SetType α, SetType β, SetType γ, SetType δ, SetType 'e, SetType 'f  ⇒
(SetType (α * β * γ * δ * 'e * 'f))
  let setElemCompare =  sextupleCompare setElemCompare setElemCompare setElemCompare setElemCompare setElemCompare setElemC
end
```

3 Function

```
(*****************************************************************************
(* A library for common operations on functions                              *)
(*****************************************************************************

open import Bool Basic_classes

declare {isabelle, hol, ocaml, coq} rename module = lem_function

open import {coq} Program.Basics

(* ------------------------- *)
(* identity function         *)
(* ------------------------- *)

val id : ∀ α. α → α
let id x = x

let inline {coq} id x = x
declare isabelle target_rep function id = 'id'
declare hol target_rep function id = 'I'


(* ------------------------- *)
(* constant function         *)
(* ------------------------- *)

val const : ∀ α β. α → β → α
let inline const x y = x

declare coq target_rep function const = 'const'
declare hol target_rep function const = 'K'


(* ------------------------- *)
(* function composition      *)
(* ------------------------- *)

val comb : ∀ α β γ. (β → γ) → (α → β) → (α → γ)
let comb f g = (fun x → f (g x))

declare coq target_rep function comb = 'compose'
declare isabelle target_rep function comb = infix 'o'
declare hol target_rep function comb = infix 'o'


(* ------------------------- *)
(* function application      *)
(* ------------------------- *)

val $ [apply] : ∀ α β. (α → β) → (α → β)
let $ f = (fun x → f x)

declare coq target_rep function apply = 'apply'
let inline {isabelle, ocaml, hol} apply f x = f x

val $> [rev_apply] : ∀ α β. α → (α → β) → β
```

```
let $> x f = f x
let inline {isabelle, ocaml, hol} rev_apply x f = f x

(* ------------------------ *)
(* flipping argument order *)
(* ------------------------ *)

val flip : ∀ α β γ. (α → β → γ) → (β → α → γ)
let flip f = (fun x y → f y x)

declare coq target_rep function flip = 'flip'
let inline {isabelle} flip f x y = f y x
declare hol target_rep function flip = 'combin$C'


(* currying / uncurrying *)

val curry : ∀ α β γ. ((α * β) → γ) → α → β → γ
let curry f = (fun a b → f (a, b))

val uncurry : ∀ α β γ. (α → β → γ) → (α * β → γ)
let uncurry f (a, b) = f a b
```

4 Maybe

```
(*****************************************************************************
(* A library for option                                                      *)
(*                                                                           *)
(* It mainly follows the Haskell Maybe - library                             *)
(*****************************************************************************

declare {hol, isabelle, ocaml, coq} rename module = lem_maybe

open import Bool Basic_classes Function

(* ========================================================================== *)
(* Basic stuff                                                               *)
(* ========================================================================== *)

type MAYBE α =
  | NOTHING
  | JUST of α

declare hol target_rep type MAYBE α = 'option' α
declare isabelle target_rep type MAYBE α = 'option' α
declare coq target_rep type MAYBE α = 'option' α
declare ocaml target_rep type MAYBE α = 'option' α

declare hol target_rep function Just = 'SOME'
declare ocaml target_rep function Just = 'Some'
declare isabelle target_rep function Just = 'Some'
declare coq target_rep function Just = 'Some'

declare hol target_rep function Nothing = 'NONE'
declare ocaml target_rep function Nothing = 'None'
declare isabelle target_rep function Nothing = 'None'
declare coq target_rep function Nothing = 'None'


val maybeEqual : ∀ α. Eq α ⇒ MAYBE α → MAYBE α → BOOL
val maybeEqualBy : ∀ α. (α → α → BOOL) → MAYBE α → MAYBE α → BOOL

let maybeEqualBy eq x y = match (x, y) with
  | (Nothing, Nothing) → true
  | (Nothing, Just _) → false
  | (Just _, Nothing) → false
  | (Just x', Just y') → (eq x' y')
end
let inline maybeEqual = maybeEqualBy (=)

declare ocaml target_rep function maybeEqualBy = 'Lem.option_equal'
let inline {hol, isabelle} maybeEqual = unsafe_structural_equality

instance ∀ α. Eq α ⇒ (Eq (MAYBE α))
  let = = maybeEqual
  let <> x y = ¬ (maybeEqual x y)
end

assert maybe_eq1 : ((Nothing : MAYBE BOOL) = Nothing)
assert maybe_eq2 : ((Just true) ≠ Nothing)
assert maybe_eq3 : ((Just false) ≠ (Just true))
```

```
assert maybe_eq4 : ((Just false) = (Just false))


let maybeCompare cmp x y = match (x, y) with
  | (Nothing, Nothing) → EQ
  | (Nothing, Just _) → LT
  | (Just _, Nothing) → GT
  | (Just x', Just y') → cmp x' y'
end

instance ∀ α. SetType α ⇒ (SetType (MAYBE α))
  let setElemCompare = maybeCompare setElemCompare
end

instance ∀ α. Ord α ⇒ (Ord (MAYBE α))
    let compare = maybeCompare compare
    let < = fun m1 → (fun m2 → maybeCompare compare m1 m2 = LT)
    let <= = fun m1 → (fun m2 → (let r = maybeCompare compare m1 m2 in r = LT ∨ r = EQ))
    let > = fun m1 → (fun m2 → maybeCompare compare m1 m2 = GT)
    let >= = fun m1 → (fun m2 → (let r = maybeCompare compare m1 m2 in r = GT ∨ r = EQ))
end

(* ---------------------------------- *)
(* maybe                              *)
(* ---------------------------------- *)

val maybe : ∀ α β. β → (α → β) → MAYBE α → β
let maybe d f mb = match mb with
  | Just a → f a
  | Nothing → d
end

declare ocaml target_rep function maybe = 'Lem.option_case'
declare isabelle target_rep function maybe = 'case_option'
declare hol target_rep function maybe d f mb = 'option_CASE' mb d f

assert maybe1 : (maybe true (fun b → ¬ b) Nothing = true)
assert maybe2 : (maybe false (fun b → ¬ b) Nothing = false)
assert maybe3 : (maybe true (fun b → ¬ b) (Just true) = false)
assert maybe4 : (maybe true (fun b → ¬ b) (Just false) = true)

(* ---------------------------------- *)
(* isJust / isNothing                 *)
(* ---------------------------------- *)

val isJust : ∀ α. MAYBE α → BOOL
let isJust mb = match mb with
  | Just _ → true
  | Nothing → false
end

declare hol target_rep function isJust = 'IS_SOME'
declare ocaml target_rep function isJust = 'Lem.is_some'
declare isabelle target_rep function isJust x = '$\neg$' (unsafe_structural_equality x Nothing)

assert isJust1 : (isJust (Just true))
assert isJust2 : (¬ (isJust (Nothing : MAYBE BOOL)))
```

```
val isNothing : ∀ α. MAYBE α → BOOL
let isNothing mb = match mb with
  | Just _ → false
  | Nothing → true
end

declare hol target_rep function isNothing = 'IS_NONE'
declare ocaml target_rep function isNothing = 'Lem.is_none'
declare isabelle target_rep function isNothing x = (unsafe_structural_equality x Nothing)

assert isNothing_1 : (¬ (isNothing (Just true)))
assert isNothing_2 : (isNothing (Nothing : MAYBE BOOL))

lemma isJustNothing : (
  (∀ x. isNothing x = ¬ (isJust x)) ∧
  (∀ v. isJust (Just v)) ∧
  (isNothing Nothing))

(* ---------------------------------- *)
(* fromMaybe                          *)
(* ---------------------------------- *)

val fromMaybe : ∀ α. α → MAYBE α → α
let fromMaybe d mb = match mb with
  | Just v → v
  | Nothing → d
end

declare ocaml target_rep function fromMaybe = 'Lem.option_default'
let inline {isabelle, hol} fromMaybe d = maybe d id

lemma fromMaybe : (
  (∀ d v. fromMaybe d (Just v) = v) ∧
  (∀ d. fromMaybe d Nothing = d))

assert fromMaybe_1 : (fromMaybe true Nothing = true)
assert fromMaybe_2 : (fromMaybe false Nothing = false)
assert fromMaybe_3 : (fromMaybe true (Just true) = true)
assert fromMaybe_4 : (fromMaybe true (Just false) = false)

(* ---------------------------------- *)
(* map                                *)
(* ---------------------------------- *)

val map : ∀ α β. (α → β) → MAYBE α → MAYBE β
let map f = maybe Nothing (fun v → Just (f v))

declare hol target_rep function map = 'OPTION_MAP'
declare ocaml target_rep function map = 'Lem.option_map'
declare isabelle target_rep function map = 'map_option'
declare coq target_rep function map = 'option_map'

lemma maybe_map : (
  (∀ f. map f Nothing = Nothing) ∧
  (∀ f v. map f (Just v) = Just (f v)))

assert map_1 : (map (fun b → ¬ b) Nothing = Nothing)
assert map_2 : (map (fun b → ¬ b) (Just true) = Just false)
```

```
assert map_3 : (map (fun b → ¬ b) (Just false) = Just true)


(* ---------------------------------- *)
(* bind                               *)
(* ---------------------------------- *)

val bind : ∀ α β. MAYBE α → (α → MAYBE β) → MAYBE β
let bind mb f = maybe Nothing f mb

declare isabelle target_rep function bind = 'Option.bind'
declare ocaml target_rep function bind = 'Lem.option_bind'
declare hol target_rep function bind = 'OPTION_BIND'

lemma maybe_bind : (
  (∀ f. bind Nothing f = Nothing) ∧
  (∀ f v. bind (Just v) f = (f v)))

assert bind_1 : (bind Nothing (fun b → Just (¬ b)) = Nothing)
assert bind_2 : (bind (Just true) (fun b → Just (¬ b)) = Just false)
assert bind_3 : (bind (Just false) (fun b → Just (¬ b)) = Just true)
assert bind_4 : (bind (Just false) (fun b → (Nothing : MAYBE BOOL)) = Nothing)
```

5 Num

```
(*****************************************************************************
(* A library for numbers                                                     *)
(*                                                                           *)
(* It mainly follows the Haskell Maybe - library                             *)
(*****************************************************************************

(* rename module to clash with existing list modules of targets  problem: renaming from inside the module i

declare {isabelle, ocaml, hol, coq} rename module = lem_num

open import Bool Basic_classes
open import {isabelle} HOL - Word.Word Complex_Main
open import {hol} integerTheory intReduce wordsTheory wordsLib ratTheory realTheory intrealTheory transcTheory

open import {coq} Coq.Numbers.BinNums Coq.ZArith.BinInt Coq.ZArith.Zpower Coq.ZArith.Zdiv Coq.ZArith.Zmax Coq.


(* ========================================================================= *)
(* Numerals                                                                  *)
(* ========================================================================= *)

(* Numerals like 0, 1, 2, 42, 4543 are built-in. That's the only use of numerals. The following type-class

declare hol target_rep type NUMERAL = 'num'
declare coq target_rep type NUMERAL = 'nat'
declare ocaml target_rep type NUMERAL = 'Nat_big_num.num'

class inline ( Numeral α )
  val fromNumeral : NUMERAL → α
end

(* ========================================================================= *)
(* Syntactic type-classes for common operations                              *)
(* ========================================================================= *)

(* Typeclasses can be used as a mean to overload constants like "+", "-", etc *)

class ( NumNegate α )
  val ~ [numNegate] : α → α
end
declare tex target_rep function numNegate = '$-$'

class ( NumAbs α )
  val abs : α → α
end

class ( NumAdd α )
  val + [numAdd] : α → α → α
end
declare tex target_rep function numAdd = infix '$+$'

class ( NumMinus α )
  val - [numMinus] : α → α → α
end
declare tex target_rep function numMinus = infix '$-$'
```

```
class ( NumMult α )
  val * [numMult] : α → α → α
end
declare tex target_rep function numMult = infix '$*$'

class ( NumPow α )
  val ** [numPow] : α → NAT → α
end
declare tex target_rep function numPow n m = special "{%e}^{%e}" n m

class ( NumDivision α )
  val / [numDivision] : α → α → α
end

class ( NumIntegerDivision α )
  val div [numIntegerDivision] : α → α → α
end


class ( NumRemainder α )
  val mod [numRemainder] : α → α → α
end

class ( NumSucc α )
  val succ : α → α
end

class ( NumPred α )
  val pred : α → α
end

(* ========================================================================== *)
(* Basic number types                                                         *)
(* ========================================================================== *)

(* ---------------------------------------- *)
(* nat                                      *)
(* ---------------------------------------- *)

(* bounded size natural numbers, i.e. positive integers *)

(* "nat" is the old type "num". It represents natural numbers.  These numbers might be bounded, however no che

declare hol target_rep type NAT = 'num'
declare isabelle target_rep type NAT = 'nat'
declare coq target_rep type NAT = 'nat'
declare ocaml target_rep type NAT = 'int'


(* ---------------------------------------- *)
(* natural                                  *)
(* ---------------------------------------- *)

(* unbounded size natural numbers *)
type NATURAL
declare hol target_rep type $\MATHBB{N}$ = 'num'
declare isabelle target_rep type $\MATHBB{N}$ = 'nat'
declare coq target_rep type $\MATHBB{N}$ = 'nat'
```

```
declare ocaml target_rep type $\MATHBB{N}$ = 'Nat_big_num.num'
declare tex target_rep type $\MATHBB{N}$ = '$\mathbb{N}$'


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* int                *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* bounded size integers with uncertain length *)

type INT
declare ocaml target_rep type INT = 'int'
declare isabelle target_rep type INT = 'int'
declare hol target_rep type INT = 'int'
declare coq target_rep type INT = 'Z'


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* integer            *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* unbounded size integers *)

type INTEGER
declare ocaml target_rep type $\MATHBB{Z}$ = 'Nat_big_num.num'
declare isabelle target_rep type $\MATHBB{Z}$ = 'int'
declare hol target_rep type $\MATHBB{Z}$ = 'int'
declare coq target_rep type $\MATHBB{Z}$ = 'Z'
declare tex target_rep type $\MATHBB{Z}$ = '$\mathbb{Z}$'

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* bint               *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* TODO the bounded ints are only partially implemented, use with care. *)

(* 32 bit integers *)
type INT32
declare ocaml target_rep type INT32 = 'Int32.t'
declare coq target_rep type INT32 = 'Z' (* ??? : better type for this in Coq? *)
declare isabelle target_rep type INT32 = 'word' 32
declare hol target_rep type INT32 = 'word'32

(* 64 bit integers *)
type INT64
declare ocaml target_rep type INT64 = 'Int64.t'
declare coq target_rep type INT64 = 'Z' (* ??? : better type for this in Coq? *)
declare isabelle target_rep type INT64 = 'word' 64
declare hol target_rep type INT64 = 'word'64


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* rational           *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* unbounded size and precision rational numbers *)

type RATIONAL
```

```
declare ocaml target_rep type RATIONAL = 'Rational.t'
declare coq target_rep type RATIONAL = 'Q' (* ??? : better type for this in Coq? *)
declare isabelle target_rep type RATIONAL = 'rat'
declare hol target_rep type RATIONAL = 'rat' (* ??? : better type for this in HOL? *)


(* -------------------------- *)
(* real                       *)
(* -------------------------- *)

(* real numbers *)
(* Note that for OCaml, this is mapped to floats with 64 bits. *)

type REAL
declare ocaml target_rep type REAL = 'float'
declare coq target_rep type REAL = 'R' (* ??? : better type for this in Coq? *)
declare isabelle target_rep type REAL = 'real'
declare hol target_rep type REAL = 'real' (* ??? : better type for this in HOL? *)


(* -------------------------- *)
(* double                     *)
(* -------------------------- *)

(* double precision floating point (64 bits) *)

type FLOAT64
declare ocaml target_rep type FLOAT64 = 'double'
declare coq target_rep type FLOAT64 = 'Q' (* ??? : better type for this in Coq? *)
declare isabelle target_rep type FLOAT64 = '???' (* ??? : better type for this in Isa? *)
declare hol target_rep type FLOAT64 = 'XXX' (* ??? : better type for this in HOL? *)

type FLOAT32
declare ocaml target_rep type FLOAT32 = 'float'
declare coq target_rep type FLOAT32 = 'Q' (* ??? : better type for this in Coq? *)
declare isabelle target_rep type FLOAT32 = '???' (* ??? : better type for this in Isa? *)
declare hol target_rep type FLOAT32 = 'XXX' (* ??? : better type for this in HOL? *)


(* ========================================================================== *)
(* Binding the standard operations for the number types                     *)
(* ========================================================================== *)


(* -------------------------- *)
(* nat                        *)
(* -------------------------- *)

val natFromNumeral : NUMERAL → NAT
declare hol target_rep function natFromNumeral x = (''x : NAT)
declare ocaml target_rep function natFromNumeral = 'Nat_big_num.to_int'
declare isabelle target_rep function natFromNumeral n = (''n : NAT)
declare coq target_rep function natFromNumeral = ''

instance (Numeral NAT)
  let fromNumeral n = natFromNumeral n
end
```

```
val natEq : nat → nat → bool
let inline natEq = unsafe_structural_equality
declare coq target_rep function natEq = 'beq_nat'
instance (Eq nat)
  let = = natEq
  let <> n1 n2 = ¬ (natEq n1 n2)
end

val natLess : nat → nat → bool
val natLessEqual : nat → nat → bool
val natGreater : nat → nat → bool
val natGreaterEqual : nat → nat → bool

declare hol target_rep function natLess = infix '<'
declare ocaml target_rep function natLess = infix '<'
declare isabelle target_rep function natLess = infix '<'
declare coq target_rep function natLess = 'nat_ltb'

declare hol target_rep function natLessEqual = infix '<='
declare ocaml target_rep function natLessEqual = infix '<='
declare isabelle target_rep function natLessEqual = infix '\<le>'
declare coq target_rep function natLessEqual = 'nat_lteb'

declare hol target_rep function natGreater = infix '>'
declare ocaml target_rep function natGreater = infix '>'
declare isabelle target_rep function natGreater = infix '>'
declare coq target_rep function natGreater = 'nat_gtb'

declare hol target_rep function natGreaterEqual = infix '>='
declare ocaml target_rep function natGreaterEqual = infix '>='
declare isabelle target_rep function natGreaterEqual = infix '\<ge>'
declare coq target_rep function natGreaterEqual = 'nat_gteb'

val natCompare : nat → nat → ordering
let inline natCompare = defaultCompare
let inline {coq, hol, isabelle} natCompare = genericCompare natLess natEq

instance (Ord nat)
  let compare = natCompare
  let < = natLess
  let <= = natLessEqual
  let > = natGreater
  let >= = natGreaterEqual
end

instance (SetType nat)
  let setElemCompare = natCompare
end

val natAdd : nat → nat → nat
declare hol target_rep function natAdd = infix '+'
declare ocaml target_rep function natAdd = infix '+'
declare isabelle target_rep function natAdd = infix '+'
declare coq target_rep function natAdd = 'Coq.Init.Peano.plus'

instance (NumAdd nat)
  let + = natAdd
end
```

```
val natMinus : NAT → NAT → NAT
declare hol target_rep function natMinus = infix '-'
declare ocaml target_rep function natMinus = 'Nat_num.nat_monus'
declare isabelle target_rep function natMinus = infix '-'
declare coq target_rep function natMinus = 'Coq.Init.Peano.minus'

instance (NumMinus NAT)
  let − = natMinus
end

val natSucc : NAT → NAT
let natSucc n = n + 1
declare hol target_rep function natSucc = 'SUC'
declare isabelle target_rep function natSucc = 'Suc'
declare ocaml target_rep function natSucc = 'succ'
declare coq target_rep function natSucc = 'S'
instance (NumSucc NAT)
  let succ = natSucc
end

val natPred : NAT → NAT
let inline natPred n = n − 1
declare hol target_rep function natPred = 'PRE'
declare ocaml target_rep function natPred = 'Nat_num.nat_pred'
declare coq target_rep function natPred = 'Coq.Init.Peano.pred'
instance (NumPred NAT)
  let pred = natPred
end

val natMult : NAT → NAT → NAT
declare hol target_rep function natMult = infix '*'
declare ocaml target_rep function natMult = infix '*'
declare isabelle target_rep function natMult = infix '*'
declare coq target_rep function natMult = 'Coq.Init.Peano.mult'

instance (NumMult NAT)
  let ∗ = natMult
end

val natDiv : NAT → NAT → NAT
declare hol target_rep function natDiv = infix 'DIV'
declare ocaml target_rep function natDiv = infix '/'
declare isabelle target_rep function natDiv = infix 'div'
declare coq target_rep function natDiv = 'Coq.Numbers.Natural.Peano.NPeano.div'

instance ( NumIntegerDivision NAT )
  let div = natDiv
end

instance ( NumDivision NAT )
  let / = natDiv
end

val natMod : NAT → NAT → NAT
declare hol target_rep function natMod = infix 'MOD'
declare ocaml target_rep function natMod = infix 'mod'
declare isabelle target_rep function natMod = infix 'mod'
```

```
declare coq target_rep function natMod = 'Coq.Numbers.Natural.Peano.NPeano.modulo'

instance ( NumRemainder NAT )
  let mod = natMod
end


val gen_pow_aux : ∀ α. (α → α → α) → α → α → NAT → α
let rec gen_pow_aux (mul : α → α → α) (a : α) (b : α) (e : NAT) =
    match e with
    | 0 → a (* cannot happen, call discipline guarentees e >= 1 *)
    | 1 → mul a b
    | (e' + 2) → let e'' = e / 2 in
                 let a' = (if (e mod 2) = 0 then a else mul a b) in
                 gen_pow_aux mul a' (mul b b) e''
  end
declare termination_argument gen_pow_aux = automatic

declare coq target_rep function gen_pow_aux = 'gen_pow_aux'

let gen_pow (one : α) (mul : α → α → α) (b : α) (e : NAT) : α =
   if e < 0 then one else
  if (e = 0) then one else gen_pow_aux mul one b e

val natPow : NAT → NAT → NAT
let {ocaml} natPow = gen_pow 1 natMult

declare hol target_rep function natPow = infix '**'
declare isabelle target_rep function natPow = infix '^'
declare coq target_rep function natPow = 'nat_power'

instance ( NumPow NAT )
  let ** = natPow
end

val natMin : NAT → NAT → NAT
let inline natMin = defaultMin
declare ocaml target_rep function natMin = 'min'
declare isabelle target_rep function natMin = 'min'
declare hol target_rep function natMin = 'MIN'
declare coq target_rep function natMin = 'nat_min'

val natMax : NAT → NAT → NAT
let inline natMax = defaultMax
declare isabelle target_rep function natMax = 'max'
declare ocaml target_rep function natMax = 'max'
declare hol target_rep function natMax = 'MAX'
declare coq target_rep function natMax = 'nat_max'

instance ( OrdMaxMin NAT )
  let max = natMax
  let min = natMin
end


(* ------------------------------------------------------------ *)
(* natural                                                      *)
(* ------------------------------------------------------------ *)
```

```
val naturalFromNumeral : NUMERAL → $\MATHBB{N}$
declare hol target_rep function naturalFromNumeral x = (''x : $\MATHBB{N}$)
declare ocaml target_rep function naturalFromNumeral = ''
declare isabelle target_rep function naturalFromNumeral n = (''n : $\MATHBB{N}$)
declare coq target_rep function naturalFromNumeral = ''

instance (Numeral $\MATHBB{N}$)
  let fromNumeral n = naturalFromNumeral n
end

val naturalEq : $\MATHBB{N}$ → $\MATHBB{N}$ → BOOL
let inline naturalEq = unsafe_structural_equality
declare ocaml target_rep function naturalEq = 'Nat_big_num.equal'
declare coq target_rep function naturalEq = 'beq_nat'
instance (Eq $\MATHBB{N}$)
  let = = naturalEq
  let <> n1 n2 = ¬ (naturalEq n1 n2)
end

val naturalLess : $\MATHBB{N}$ → $\MATHBB{N}$ → BOOL
val naturalLessEqual : $\MATHBB{N}$ → $\MATHBB{N}$ → BOOL
val naturalGreater : $\MATHBB{N}$ → $\MATHBB{N}$ → BOOL
val naturalGreaterEqual : $\MATHBB{N}$ → $\MATHBB{N}$ → BOOL

declare hol target_rep function naturalLess = infix '<'
declare ocaml target_rep function naturalLess = 'Nat_big_num.less'
declare isabelle target_rep function naturalLess = infix '<'
declare coq target_rep function naturalLess = 'nat_ltb'

declare hol target_rep function naturalLessEqual = infix '<='
declare ocaml target_rep function naturalLessEqual = 'Nat_big_num.less_equal'
declare isabelle target_rep function naturalLessEqual = infix '\<le>'
declare coq target_rep function naturalLessEqual = 'nat_lteb'

declare hol target_rep function naturalGreater = infix '>'
declare ocaml target_rep function naturalGreater = 'Nat_big_num.greater'
declare isabelle target_rep function naturalGreater = infix '>'
declare coq target_rep function naturalGreater = 'nat_gtb'

declare hol target_rep function naturalGreaterEqual = infix '>='
declare ocaml target_rep function naturalGreaterEqual = 'Nat_big_num.greater_equal'
declare isabelle target_rep function naturalGreaterEqual = infix '\<ge>'
declare coq target_rep function naturalGreaterEqual = 'nat_gteb'

val naturalCompare : $\MATHBB{N}$ → $\MATHBB{N}$ → ORDERING
let inline naturalCompare = defaultCompare
let inline {coq, isabelle, hol} naturalCompare = genericCompare naturalLess naturalEq
declare ocaml target_rep function naturalCompare = 'Nat_big_num.compare'

instance (Ord $\MATHBB{N}$)
  let compare = naturalCompare
  let < = naturalLess
  let <= = naturalLessEqual
  let > = naturalGreater
  let >= = naturalGreaterEqual
end
```

```
instance (SetType $\MATHBB{N}$)
  let setElemCompare =  naturalCompare
end

val naturalAdd : $\MATHBB{N}$ → $\MATHBB{N}$ → $\MATHBB{N}$
declare hol target_rep function naturalAdd = infix '+'
declare ocaml target_rep function naturalAdd = 'Nat_big_num.add'
declare isabelle target_rep function naturalAdd = infix '+'
declare coq target_rep function naturalAdd = 'Coq.Init.Peano.plus'

instance (NumAdd $\MATHBB{N}$)
  let + =  naturalAdd
end

val naturalMinus : $\MATHBB{N}$ → $\MATHBB{N}$ → $\MATHBB{N}$
declare hol target_rep function naturalMinus = infix '-'
declare ocaml target_rep function naturalMinus = 'Nat_big_num.sub_nat'
declare isabelle target_rep function naturalMinus = infix '-'
declare coq target_rep function naturalMinus = 'Coq.Init.Peano.minus'

instance (NumMinus $\MATHBB{N}$)
  let − =  naturalMinus
end

val naturalSucc : $\MATHBB{N}$ → $\MATHBB{N}$
let naturalSucc n =  n + 1
declare hol target_rep function naturalSucc = 'SUC'
declare isabelle target_rep function naturalSucc = 'Suc'
declare ocaml target_rep function naturalSucc = 'Nat_big_num.succ'
declare coq target_rep function naturalSucc = 'S'
instance (NumSucc $\MATHBB{N}$)
  let succ =  naturalSucc
end

val naturalPred : $\MATHBB{N}$ → $\MATHBB{N}$
let inline naturalPred n = n − 1
declare hol target_rep function naturalPred = 'PRE'
declare ocaml target_rep function naturalPred = 'Nat_big_num.pred_nat'
declare coq target_rep function naturalPred = 'Coq.Init.Peano.pred'
instance (NumPred $\MATHBB{N}$)
  let pred =  naturalPred
end

val naturalMult : $\MATHBB{N}$ → $\MATHBB{N}$ → $\MATHBB{N}$
declare hol target_rep function naturalMult = infix '*'
declare ocaml target_rep function naturalMult = 'Nat_big_num.mul'
declare isabelle target_rep function naturalMult = infix '*'
declare coq target_rep function naturalMult = 'Coq.Init.Peano.mult'

instance (NumMult $\MATHBB{N}$)
  let ∗ =  naturalMult
end


val naturalPow : $\MATHBB{N}$ → NAT → $\MATHBB{N}$
declare hol target_rep function naturalPow = infix '**'
declare ocaml target_rep function naturalPow = 'Nat_big_num.pow_int'
declare isabelle target_rep function naturalPow = infix '^'
```

```
declare coq target_rep function naturalPow = 'nat_power'

instance ( NumPow $\mathbb{N}$ )
  let ** = naturalPow
end

val naturalDiv : $\mathbb{N}$ → $\mathbb{N}$ → $\mathbb{N}$
declare hol target_rep function naturalDiv = infix 'DIV'
declare ocaml target_rep function naturalDiv = 'Nat_big_num.div'
declare isabelle target_rep function naturalDiv = infix 'div'
declare coq target_rep function naturalDiv = 'Coq.Numbers.Natural.Peano.NPeano.div'

instance ( NumIntegerDivision $\mathbb{N}$ )
  let div = naturalDiv
end

instance ( NumDivision $\mathbb{N}$ )
  let / = naturalDiv
end

val naturalMod : $\mathbb{N}$ → $\mathbb{N}$ → $\mathbb{N}$
declare hol target_rep function naturalMod = infix 'MOD'
declare ocaml target_rep function naturalMod = 'Nat_big_num.modulus'
declare isabelle target_rep function naturalMod = infix 'mod'
declare coq target_rep function naturalMod = 'Coq.Numbers.Natural.Peano.NPeano.modulo'

instance ( NumRemainder $\mathbb{N}$ )
  let mod = naturalMod
end

val naturalMin : $\mathbb{N}$ → $\mathbb{N}$ → $\mathbb{N}$
let inline naturalMin = defaultMin
declare isabelle target_rep function naturalMin = 'min'
declare ocaml target_rep function naturalMin = 'Nat_big_num.min'
declare hol target_rep function naturalMin = 'MIN'
declare coq target_rep function naturalMin = 'nat_min'

val naturalMax : $\mathbb{N}$ → $\mathbb{N}$ → $\mathbb{N}$
let inline naturalMax = defaultMax
declare isabelle target_rep function naturalMax = 'max'
declare ocaml target_rep function naturalMax = 'Nat_big_num.max'
declare hol target_rep function naturalMax = 'MAX'
declare coq target_rep function naturalMax = 'nat_max'

instance ( OrdMaxMin $\mathbb{N}$ )
  let max = naturalMax
  let min = naturalMin
end


(* ---------------------------------- *)
(* int                                *)
(* ---------------------------------- *)

val intFromNumeral : NUMERAL → INT
declare ocaml target_rep function intFromNumeral = 'Nat_big_num.to_int'
declare isabelle target_rep function intFromNumeral n = (''n : INT)
declare hol target_rep function intFromNumeral n = (''n : INT)
```

```
declare coq target_rep function intFromNumeral n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n)))

instance (Numeral INT)
  let fromNumeral n = intFromNumeral n
end

val intEq : INT → INT → BOOL
let inline intEq = unsafe_structural_equality
declare coq target_rep function intEq = 'Z.eqb'
instance (Eq INT)
  let = = intEq
  let <> n1 n2 = ¬ (intEq n1 n2)
end

val intLess : INT → INT → BOOL
val intLessEqual : INT → INT → BOOL
val intGreater : INT → INT → BOOL
val intGreaterEqual : INT → INT → BOOL

declare hol target_rep function intLess = infix '<'
declare ocaml target_rep function intLess = infix '<'
declare isabelle target_rep function intLess = infix '<'
declare coq target_rep function intLess = 'int_ltb'

declare hol target_rep function intLessEqual = infix '<='
declare ocaml target_rep function intLessEqual = infix '<='
declare isabelle target_rep function intLessEqual = infix '\<le>'
declare coq target_rep function intLessEqual = 'int_lteb'

declare hol target_rep function intGreater = infix '>'
declare ocaml target_rep function intGreater = infix '>'
declare isabelle target_rep function intGreater = infix '>'
declare coq target_rep function intGreater = 'int_gtb'

declare hol target_rep function intGreaterEqual = infix '>='
declare ocaml target_rep function intGreaterEqual = infix '>='
declare isabelle target_rep function intGreaterEqual = infix '\<ge>'
declare coq target_rep function intGreaterEqual = 'int_gteb'

val intCompare : INT → INT → ORDERING
let inline intCompare = defaultCompare
let inline {coq, isabelle, hol} intCompare = genericCompare intLess intEq
declare ocaml target_rep function intCompare = 'compare'

instance (Ord INT)
  let compare = intCompare
  let < = intLess
  let <= = intLessEqual
  let > = intGreater
  let >= = intGreaterEqual
end

instance (SetType INT)
  let setElemCompare = intCompare
end

val intNegate : INT → INT
declare hol target_rep function intNegate i = '~' i
```

```
declare ocaml target_rep function intNegate i = ('~-' i)
declare isabelle target_rep function intNegate i = '-' i
declare coq target_rep function intNegate i = ('Coq.ZArith.BinInt.Z.sub' 'Z'0 i)

instance (NumNegate INT)
  let ~ = intNegate
end

val intAbs : INT → INT
declare hol target_rep function intAbs = 'ABS'
declare ocaml target_rep function intAbs = 'abs'
declare isabelle target_rep function intAbs = 'abs'
declare coq target_rep function intAbs input = ('Z.pred' ('Z.pos' ('P_of_succ_nat' ('Z.abs_nat' input))))
(* TODO: check *)

instance (NumAbs INT)
  let abs = intAbs
end

val intAdd : INT → INT → INT
declare hol target_rep function intAdd = infix '+'
declare ocaml target_rep function intAdd = infix '+'
declare isabelle target_rep function intAdd = infix '+'
declare coq target_rep function intAdd = 'Coq.ZArith.BinInt.Z.add'

instance (NumAdd INT)
  let + = intAdd
end

val intMinus : INT → INT → INT
declare hol target_rep function intMinus = infix '-'
declare ocaml target_rep function intMinus = infix '-'
declare isabelle target_rep function intMinus = infix '-'
declare coq target_rep function intMinus = 'Coq.ZArith.BinInt.Z.sub'

instance (NumMinus INT)
  let - = intMinus
end

val intSucc : INT → INT
let inline intSucc n = n + 1
declare ocaml target_rep function intSucc = 'succ'
instance (NumSucc INT)
  let succ = intSucc
end

val intPred : INT → INT
let inline intPred n = n - 1
declare ocaml target_rep function intPred = 'pred'
instance (NumPred INT)
  let pred = intPred
end

val intMult : INT → INT → INT
declare hol target_rep function intMult = infix '*'
declare ocaml target_rep function intMult = infix '*'
declare isabelle target_rep function intMult = infix '*'
declare coq target_rep function intMult = 'Coq.ZArith.BinInt.Z.mul'
```

```
instance (NumMult INT)
  let * =  intMult
end


val intPow : INT → NAT → INT
let {ocaml} intPow =  gen_pow 1 intMult
declare hol target_rep function intPow = infix '**'
declare isabelle target_rep function intPow = infix '^'
declare coq target_rep function intPow = 'Coq.ZArith.Zpower.Zpower_nat'

instance ( NumPow INT )
  let ** =  intPow
end

val intDiv : INT → INT → INT
declare hol target_rep function intDiv = infix '/'
declare ocaml target_rep function intDiv = 'Nat_num.int_div'
declare isabelle target_rep function intDiv = infix 'div'
declare coq target_rep function intDiv = 'Z.div'

instance ( NumIntegerDivision INT )
  let div =  intDiv
end

instance ( NumDivision INT )
  let / =  intDiv
end

val intMod : INT → INT → INT
declare hol target_rep function intMod = infix '%'
declare ocaml target_rep function intMod = 'Nat_num.int_mod'
declare isabelle target_rep function intMod = infix 'mod'
declare coq target_rep function intMod = 'Coq.ZArith.Zdiv.Zmod'

instance ( NumRemainder INT )
  let mod =  intMod
end

val intMin : INT → INT → INT
let inline intMin = defaultMin
declare isabelle target_rep function intMin = 'min'
declare ocaml target_rep function intMin = 'min'
declare hol target_rep function intMin = 'int_min'
declare coq target_rep function intMin = 'Z.min'

val intMax : INT → INT → INT
let inline intMax = defaultMax
declare isabelle target_rep function intMax = 'max'
declare ocaml target_rep function intMax = 'max'
declare hol target_rep function intMax = 'int_max'
declare coq target_rep function intMax = 'Z.max'

instance ( OrdMaxMin INT )
  let max =  intMax
  let min =  intMin
end
```

```
(* ------------------------- *)
(* int32                     *)
(* ------------------------- *)
val int32FromNumeral : NUMERAL → INT32

declare ocaml target_rep function int32FromNumeral = 'Nat_big_num.to_int'32
declare isabelle target_rep function int32FromNumeral n = (('word_of_int' n) : INT32)
declare hol target_rep function int32FromNumeral n = (('n2w' n) : INT32)
declare coq target_rep function int32FromNumeral n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))) (* TODO: check *)

instance (Numeral INT32)
  let fromNumeral n = int32FromNumeral n
end

val int32Eq : INT32 → INT32 → BOOL
let inline int32Eq = unsafe_structural_equality
declare coq target_rep function int32Eq = 'Z.eqb'

instance (Eq INT32)
  let = = int32Eq
  let <> n1 n2 = ¬ (int32Eq n1 n2)
end

val int32Less : INT32 → INT32 → BOOL
val int32LessEqual : INT32 → INT32 → BOOL
val int32Greater : INT32 → INT32 → BOOL
val int32GreaterEqual : INT32 → INT32 → BOOL

declare ocaml target_rep function int32Less = infix '<'
declare isabelle target_rep function int32Less = 'word_sless'
declare hol target_rep function int32Less = infix '<'
(*TODO: Implement the following correctly.*)
declare coq target_rep function int32Less = 'int_ltb'

declare ocaml target_rep function int32LessEqual = infix '<='
declare isabelle target_rep function int32LessEqual = 'word_sle'
declare hol target_rep function int32LessEqual = infix '<='
(*TODO: Implement the following correctly.*)
declare coq target_rep function int32LessEqual = 'int_lteb'

declare ocaml target_rep function int32Greater = infix '>'
let inline {isabelle} int32Greater x y = int32Less y x
declare hol target_rep function int32Greater = infix '>'
(*TODO: Implement the following correctly.*)
declare coq target_rep function int32Greater = 'int_gtb'

declare ocaml target_rep function int32GreaterEqual = infix '>='
let inline {isabelle} int32GreaterEqual x y = int32LessEqual y x
declare hol target_rep function int32GreaterEqual = infix '>='
(*TODO: Implement the following correctly.*)
declare coq target_rep function int32GreaterEqual = 'int_gteb'

val int32Compare : INT32 → INT32 → ORDERING
let inline int32Compare = defaultCompare
let inline {coq, isabelle, hol} int32Compare = genericCompare int32Less int32Eq
declare ocaml target_rep function int32Compare = 'Int32.compare'
```

```
instance (Ord INT32)
  let compare = int32Compare
  let < = int32Less
  let <= = int32LessEqual
  let > = int32Greater
  let >= = int32GreaterEqual
end

instance (SetType INT32)
  let setElemCompare = int32Compare
end

val int32Negate : INT32 → INT32
declare ocaml target_rep function int32Negate = 'Int32.neg'
declare isabelle target_rep function int32Negate i = '-' i
declare hol target_rep function int32Negate i = (('-' i) : INT32)
(*TODO: Implement the following correctly.*)
declare coq target_rep function int32Negate i = ('Coq.ZArith.BinInt.Z.sub' 'Z'0 i)

instance (NumNegate INT32)
  let ~ = int32Negate
end

val int32Abs : INT32 → INT32
let int32Abs i = (if 0 ≤ i then i else −i)
declare ocaml target_rep function int32Abs = 'Int32.abs'

instance (NumAbs INT32)
  let abs = int32Abs
end


val int32Add : INT32 → INT32 → INT32
declare ocaml target_rep function int32Add = 'Int32.add'
declare isabelle target_rep function int32Add = infix '+'
(*TODO: Implement the following two correctly.*)
declare hol target_rep function int32Add i1 i2 = (('word_add' i1 i2) : INT32)
declare coq target_rep function int32Add = 'Coq.ZArith.BinInt.Z.add'

instance (NumAdd INT32)
  let + = int32Add
end

val int32Minus : INT32 → INT32 → INT32
declare ocaml target_rep function int32Minus = 'Int32.sub'
declare isabelle target_rep function int32Minus = infix '-'
(*TODO: Implement the following two correctly.*)
declare hol target_rep function int32Minus i1 i2 = (('word_sub' i1 i2) : INT32)
declare coq target_rep function int32Minus = 'Coq.ZArith.BinInt.Z.sub'

instance (NumMinus INT32)
  let − = int32Minus
end

val int32Succ : INT32 → INT32
let inline int32Succ n = n + 1
declare ocaml target_rep function int32Succ = 'Int32.succ'
```

```
instance (NumSucc INT32)
  let succ =  int32Succ
end

val int32Pred : INT32 → INT32
let inline int32Pred n = n − 1
declare ocaml target_rep function int32Pred = 'Int32.pred'
instance (NumPred INT32)
  let pred =  int32Pred
end

val int32Mult : INT32 → INT32 → INT32
declare ocaml target_rep function int32Mult = 'Int32.mul'
declare isabelle target_rep function int32Mult = infix '*'
declare hol target_rep function int32Mult i1 i2 = (('word_mul' i1 i2) : INT32)
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32Mult = 'Coq.ZArith.BinInt.Z.mul'

instance (NumMult INT32)
  let * =  int32Mult
end


val int32Pow : INT32 → NAT → INT32
let {ocaml, hol} int32Pow =  gen_pow 1 int32Mult
declare isabelle target_rep function int32Pow = infix '^'
(*TODO: Implement the following two correctly. *)
declare coq target_rep function int32Pow = 'Coq.ZArith.Zpower.Zpower_nat'

instance ( NumPow INT32 )
  let ** =  int32Pow
end

val int32Div : INT32 → INT32 → INT32
declare ocaml target_rep function int32Div = 'Nat_num.int32_div'
declare isabelle target_rep function int32Div = infix 'div'
declare hol target_rep function int32Div i1 i2 = (('word_div' i1 i2) : INT32)
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32Div = 'Z.div'

instance ( NumIntegerDivision INT32 )
  let div =  int32Div
end

instance ( NumDivision INT32 )
  let / =  int32Div
end

val int32Mod : INT32 → INT32 → INT32
declare ocaml target_rep function int32Mod = 'Nat_num.int32_mod'
declare isabelle target_rep function int32Mod = infix 'mod'
declare hol target_rep function int32Mod i1 i2 = (('word_mod' i1 i2) : INT32)
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32Mod = 'Coq.ZArith.Zdiv.Zmod'

instance ( NumRemainder INT32 )
  let mod =  int32Mod
end
```

```
val int32Min : INT32 → INT32 → INT32
let inline int32Min = defaultMin
declare hol target_rep function int32Min = 'word_smin'
(*TODO: Implement the following correctly.*)
declare coq target_rep function int32Min = 'Z.min'

val int32Max : INT32 → INT32 → INT32
let inline int32Max = defaultMax
declare hol target_rep function int32Max = 'word_smax'
(*TODO: Implement the following correctly.*)
declare coq target_rep function int32Max = 'Z.max'

instance ( OrdMaxMin INT32 )
  let max = int32Max
  let min = int32Min
end



(* ------------------------- *)
(* int64                     *)
(* ------------------------- *)
val int64FromNumeral : NUMERAL → INT64

declare ocaml target_rep function int64FromNumeral = 'Nat_big_num.to_int'64
declare isabelle target_rep function int64FromNumeral n = (('word_of_int' n) : INT64)
declare hol target_rep function int64FromNumeral n = (('n2w' n) : INT64)
declare coq target_rep function int64FromNumeral n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))) (* TODO: check *)

instance (Numeral INT64)
  let fromNumeral n = int64FromNumeral n
end

val int64Eq : INT64 → INT64 → BOOL
let inline int64Eq = unsafe_structural_equality
declare coq target_rep function int64Eq = 'Z.eqb'

instance (Eq INT64)
  let = = int64Eq
  let <> n1 n2 = ¬ (int64Eq n1 n2)
end

val int64Less : INT64 → INT64 → BOOL
val int64LessEqual : INT64 → INT64 → BOOL
val int64Greater : INT64 → INT64 → BOOL
val int64GreaterEqual : INT64 → INT64 → BOOL

declare ocaml target_rep function int64Less = infix '<'
declare isabelle target_rep function int64Less = 'word_sless'
declare hol target_rep function int64Less = infix '<'
(*TODO: Implement the following correctly.*)
declare coq target_rep function int64Less = 'int_ltb'

declare ocaml target_rep function int64LessEqual = infix '<='
declare isabelle target_rep function int64LessEqual = 'word_sle'
declare hol target_rep function int64LessEqual = infix '<='
(*TODO: Implement the following correctly.*)
```

```
declare coq target_rep function int64LessEqual = 'int_lteb'

declare ocaml target_rep function int64Greater = infix '>'
let inline {isabelle} int64Greater x y = int64Less y x
declare hol target_rep function int64Greater = infix '>'
(*TODO: Implement the following correctly.*)
declare coq target_rep function int64Greater = 'int_gtb'

declare ocaml target_rep function int64GreaterEqual = infix '>='
let inline {isabelle} int64GreaterEqual x y = int64LessEqual y x
declare hol target_rep function int64GreaterEqual = infix '>='
(*TODO: Implement the following correctly.*)
declare coq target_rep function int64GreaterEqual = 'int_gteb'

val int64Compare : INT64 → INT64 → ORDERING
let inline int64Compare = defaultCompare
let inline {coq, isabelle, hol} int64Compare = genericCompare int64Less int64Eq
declare ocaml target_rep function int64Compare = 'Int64.compare'

instance (Ord INT64)
  let compare = int64Compare
  let < = int64Less
  let <= = int64LessEqual
  let > = int64Greater
  let >= = int64GreaterEqual
end

instance (SetType INT64)
  let setElemCompare = int64Compare
end

val int64Negate : INT64 → INT64
declare ocaml target_rep function int64Negate = 'Int64.neg'
declare isabelle target_rep function int64Negate i = '-' i
declare hol target_rep function int64Negate i = (('-' i) : INT64)
(*TODO: Implement the following one correctly.*)
declare coq target_rep function int64Negate i = ('Coq.ZArith.BinInt.Z.sub' 'Z'0 i)

instance (NumNegate INT64)
  let ~ = int64Negate
end

val int64Abs : INT64 → INT64
let int64Abs i = (if 0 ≤ i then i else −i)
declare ocaml target_rep function int64Abs = 'Int64.abs'

instance (NumAbs INT64)
  let abs = int64Abs
end


val int64Add : INT64 → INT64 → INT64
declare ocaml target_rep function int64Add = 'Int64.add'
declare isabelle target_rep function int64Add = infix '+'
declare hol target_rep function int64Add i1 i2 = (('word_add' i1 i2) : INT64)
(*TODO: Implement the following one correctly.*)
declare coq target_rep function int64Add = 'Coq.ZArith.BinInt.Z.add'
```

```
instance (NumAdd INT64)
  let + =  int64Add
end

val int64Minus : INT64 → INT64 → INT64
declare ocaml target_rep function int64Minus = 'Int64.sub'
declare isabelle target_rep function int64Minus = infix '-'
declare hol target_rep function int64Minus i1 i2 = (('word_sub' i1 i2) : INT64)
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Minus = 'Coq.ZArith.BinInt.Z.sub'

instance (NumMinus INT64)
  let − =  int64Minus
end

val int64Succ : INT64 → INT64
let inline int64Succ n = n + 1
declare ocaml target_rep function int64Succ = 'Int64.succ'

instance (NumSucc INT64)
  let succ =  int64Succ
end

val int64Pred : INT64 → INT64
let inline int64Pred n = n − 1
declare ocaml target_rep function int64Pred = 'Int64.pred'
instance (NumPred INT64)
  let pred =  int64Pred
end

val int64Mult : INT64 → INT64 → INT64
declare ocaml target_rep function int64Mult = 'Int64.mul'
declare isabelle target_rep function int64Mult = infix '*'
declare hol target_rep function int64Mult i1 i2 = (('word_mul' i1 i2) : INT64)
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Mult = 'Coq.ZArith.BinInt.Z.mul'

instance (NumMult INT64)
  let * =  int64Mult
end


val int64Pow : INT64 → NAT → INT64
let {ocaml, hol} int64Pow =  gen_pow 1 int64Mult
declare isabelle target_rep function int64Pow = infix '^'
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Pow = 'Coq.ZArith.Zpower.Zpower_nat'

instance ( NumPow INT64 )
  let ** =  int64Pow
end

val int64Div : INT64 → INT64 → INT64
declare ocaml target_rep function int64Div = 'Nat_num.int64_div'
declare isabelle target_rep function int64Div = infix 'div'
(*TODO: Implement the following two correctly. *)
declare hol target_rep function int64Div i1 i2 = (('word_div' i1 i2) : INT64)
declare coq target_rep function int64Div = 'Z.div'
```

```
instance ( NumIntegerDivision INT64 )
  let div =  int64Div
end

instance ( NumDivision INT64 )
  let / =  int64Div
end

val int64Mod : INT64 → INT64 → INT64
declare ocaml target_rep function int64Mod = 'Nat_num.int64_mod'
declare isabelle target_rep function int64Mod = infix 'mod'
(*TODO: Implement the following two correctly. *)
declare hol target_rep function int64Mod i1 i2 = (('word_mod' i1 i2) : INT64)
declare coq target_rep function int64Mod = 'Coq.ZArith.Zdiv.Zmod'

instance ( NumRemainder INT64 )
  let mod =  int64Mod
end

val int64Min : INT64 → INT64 → INT64
let inline int64Min = defaultMin
declare hol target_rep function int64Min = 'word_smin'
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Min = 'Z.min'

val int64Max : INT64 → INT64 → INT64
let inline int64Max = defaultMax
declare hol target_rep function int64Max = 'word_smax'
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Max = 'Z.max'

instance ( OrdMaxMin INT64 )
  let max =  int64Max
  let min =  int64Min
end


(* ---------------------------------- *)
(* integer                            *)
(* ---------------------------------- *)

val integerFromNumeral : NUMERAL → $\MATHBB{Z}$
declare ocaml target_rep function integerFromNumeral = ''
declare isabelle target_rep function integerFromNumeral n = (''n : $\MATHBB{Z}$)
declare hol target_rep function integerFromNumeral n = (''n : $\MATHBB{Z}$)
declare coq target_rep function integerFromNumeral n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n)))

instance (Numeral $\MATHBB{Z}$)
  let fromNumeral n =  integerFromNumeral n
end

val integerFromNat : NAT → $\MATHBB{Z}$
declare hol target_rep function integerFromNat = 'int_of_num'
declare ocaml target_rep function integerFromNat = 'Nat_big_num.of_int'
declare isabelle target_rep function integerFromNat = 'int'
declare coq target_rep function integerFromNat n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))) (* TODO: check *)
```

```
val integerEq : $\mathbb{Z}$ → $\mathbb{Z}$ → bool
let inline integerEq = unsafe_structural_equality
declare ocaml target_rep function integerEq = 'Nat_big_num.equal'
declare coq target_rep function integerEq = 'Z.eqb'
instance (Eq $\mathbb{Z}$)
  let = = integerEq
  let <> n1 n2 = ¬ (integerEq n1 n2)
end

val integerLess : $\mathbb{Z}$ → $\mathbb{Z}$ → bool
val integerLessEqual : $\mathbb{Z}$ → $\mathbb{Z}$ → bool
val integerGreater : $\mathbb{Z}$ → $\mathbb{Z}$ → bool
val integerGreaterEqual : $\mathbb{Z}$ → $\mathbb{Z}$ → bool

declare hol target_rep function integerLess = infix '<'
declare ocaml target_rep function integerLess = 'Nat_big_num.less'
declare isabelle target_rep function integerLess = infix '<'
declare coq target_rep function integerLess = 'int_ltb'

declare hol target_rep function integerLessEqual = infix '<='
declare ocaml target_rep function integerLessEqual = 'Nat_big_num.less_equal'
declare isabelle target_rep function integerLessEqual = infix '\<le>'
declare coq target_rep function integerLessEqual = 'int_lteb'

declare hol target_rep function integerGreater = infix '>'
declare ocaml target_rep function integerGreater = 'Nat_big_num.greater'
declare isabelle target_rep function integerGreater = infix '>'
declare coq target_rep function integerGreater = 'int_gtb'

declare hol target_rep function integerGreaterEqual = infix '>='
declare ocaml target_rep function integerGreaterEqual = 'Nat_big_num.greater_equal'
declare isabelle target_rep function integerGreaterEqual = infix '\<ge>'
declare coq target_rep function integerGreaterEqual = 'int_gteb'

val integerCompare : $\mathbb{Z}$ → $\mathbb{Z}$ → ordering
let inline integerCompare = defaultCompare
let inline {coq, isabelle, hol} integerCompare = genericCompare integerLess integerEq
declare ocaml target_rep function integerCompare = 'Nat_big_num.compare'

instance (Ord $\mathbb{Z}$)
  let compare = integerCompare
  let < = integerLess
  let <= = integerLessEqual
  let > = integerGreater
  let >= = integerGreaterEqual
end

instance (SetType $\mathbb{Z}$)
  let setElemCompare = integerCompare
end

val integerNegate : $\mathbb{Z}$ → $\mathbb{Z}$
declare hol target_rep function integerNegate i = '~' i
declare ocaml target_rep function integerNegate = 'Nat_big_num.negate'
declare isabelle target_rep function integerNegate i = '-' i
declare coq target_rep function integerNegate i = ('Coq.ZArith.BinInt.Z.sub' 'Z'0 i)

instance (NumNegate $\mathbb{Z}$)
```

```
  let ~ =  integerNegate
end

val integerAbs : $\MATHBB{Z}$ → $\MATHBB{Z}$
declare hol target_rep function integerAbs = 'ABS'
declare ocaml target_rep function integerAbs = 'Nat_big_num.abs'
declare isabelle target_rep function integerAbs = 'abs'
declare coq target_rep function integerAbs input = ('Z.pred' ('Z.pos' ('P_of_succ_nat' ('Z.abs_nat' input))))
(* TODO : check *)

instance (NumAbs $\MATHBB{Z}$)
  let abs =  integerAbs
end

val integerAdd : $\MATHBB{Z}$ → $\MATHBB{Z}$ → $\MATHBB{Z}$
declare hol target_rep function integerAdd = infix '+'
declare ocaml target_rep function integerAdd = 'Nat_big_num.add'
declare isabelle target_rep function integerAdd = infix '+'
declare coq target_rep function integerAdd = 'Coq.ZArith.BinInt.Z.add'

instance (NumAdd $\MATHBB{Z}$)
  let + =  integerAdd
end

val integerMinus : $\MATHBB{Z}$ → $\MATHBB{Z}$ → $\MATHBB{Z}$
declare hol target_rep function integerMinus = infix '-'
declare ocaml target_rep function integerMinus = 'Nat_big_num.sub'
declare isabelle target_rep function integerMinus = infix '-'
declare coq target_rep function integerMinus = 'Coq.ZArith.BinInt.Z.sub'

instance (NumMinus $\MATHBB{Z}$)
  let − =  integerMinus
end

val integerSucc : $\MATHBB{Z}$ → $\MATHBB{Z}$
let inline integerSucc n = n + 1
declare ocaml target_rep function integerSucc = 'Nat_big_num.succ'
instance (NumSucc $\MATHBB{Z}$)
  let succ =  integerSucc
end

val integerPred : $\MATHBB{Z}$ → $\MATHBB{Z}$
let inline integerPred n = n − 1
declare ocaml target_rep function integerPred = 'Nat_big_num.pred'
instance (NumPred $\MATHBB{Z}$)
  let pred =  integerPred
end

val integerMult : $\MATHBB{Z}$ → $\MATHBB{Z}$ → $\MATHBB{Z}$
declare hol target_rep function integerMult = infix '*'
declare ocaml target_rep function integerMult = 'Nat_big_num.mul'
declare isabelle target_rep function integerMult = infix '*'
declare coq target_rep function integerMult = 'Coq.ZArith.BinInt.Z.mul'

instance (NumMult $\MATHBB{Z}$)
  let * =  integerMult
end
```

```
val integerPow : $\mathbb{Z}$ → nat → $\mathbb{Z}$
declare hol target_rep function integerPow = infix '**'
declare ocaml target_rep function integerPow = 'Nat_big_num.pow_int'
declare isabelle target_rep function integerPow = infix '^'
declare coq target_rep function integerPow = 'Coq.ZArith.Zpower.Zpower_nat'

instance ( NumPow $\mathbb{Z}$ )
  let ** = integerPow
end

val integerDiv : $\mathbb{Z}$ → $\mathbb{Z}$ → $\mathbb{Z}$
declare hol target_rep function integerDiv = infix '/'
declare ocaml target_rep function integerDiv = 'Nat_big_num.div'
declare isabelle target_rep function integerDiv = infix 'div'
declare coq target_rep function integerDiv = 'Z.div'

instance ( NumIntegerDivision $\mathbb{Z}$ )
  let div = integerDiv
end

instance ( NumDivision $\mathbb{Z}$ )
  let / = integerDiv
end

val integerMod : $\mathbb{Z}$ → $\mathbb{Z}$ → $\mathbb{Z}$
declare hol target_rep function integerMod = infix '%'
declare ocaml target_rep function integerMod = 'Nat_big_num.modulus'
declare isabelle target_rep function integerMod = infix 'mod'
declare coq target_rep function integerMod = 'Coq.ZArith.Zdiv.Zmod'

instance ( NumRemainder $\mathbb{Z}$ )
  let mod = integerMod
end

val integerMin : $\mathbb{Z}$ → $\mathbb{Z}$ → $\mathbb{Z}$
let inline integerMin = defaultMin
declare isabelle target_rep function integerMin = 'min'
declare ocaml target_rep function integerMin = 'Nat_big_num.min'
declare hol target_rep function integerMin = 'int_min'
declare coq target_rep function integerMin = 'Z.min'

val integerMax : $\mathbb{Z}$ → $\mathbb{Z}$ → $\mathbb{Z}$
let inline integerMax = defaultMax
declare isabelle target_rep function integerMax = 'max'
declare ocaml target_rep function integerMax = 'Nat_big_num.max'
declare hol target_rep function integerMax = 'int_max'
declare coq target_rep function integerMax = 'Z.max'

instance ( OrdMaxMin $\mathbb{Z}$ )
  let max = integerMax
  let min = integerMin
end



(* - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* rational                                             *)
```

```
(* -------------------------- *)

val rationalFromNumeral : NUMERAL → RATIONAL
declare ocaml target_rep function rationalFromNumeral n = ('Rational.of_big_int' n)
declare isabelle target_rep function rationalFromNumeral n = ('Fract' (''n : $\MATHBB{Z}$) (1 :
$\MATHBB{Z}$))
declare hol target_rep function rationalFromNumeral n = (''n : RATIONAL)
declare coq target_rep function rationalFromNumeral n = ('inject_Z' ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))))


instance (Numeral RATIONAL)
  let fromNumeral n = rationalFromNumeral n
end

val rationalFromInt : INT → RATIONAL
declare ocaml target_rep function rationalFromInt n = ('Rational.of_int' n)
declare isabelle target_rep function rationalFromInt n = ('Fract' n (1 : $\MATHBB{Z}$))
declare hol target_rep function rationalFromInt n = ('rat_of_int' n)
declare coq target_rep function rationalFromInt n = ('inject_Z' n)

val rationalFromInteger : $\MATHBB{Z}$ → RATIONAL
declare ocaml target_rep function rationalFromInteger n = ('Rational.of_big_int' n)
declare isabelle target_rep function rationalFromInteger n = ('Fract' n (1 : $\MATHBB{Z}$))
declare hol target_rep function rationalFromInteger n = ('rat_of_int' n)
declare coq target_rep function rationalFromInteger n = ('inject_Z' n)

val rationalEq : RATIONAL → RATIONAL → BOOL
let inline rationalEq = unsafe_structural_equality
declare ocaml target_rep function rationalEq = 'Rational.equal'
declare coq target_rep function rationalEq = 'Qeq_bool'
instance (Eq RATIONAL)
  let = = rationalEq
  let <> n1 n2 = ¬ (rationalEq n1 n2)
end

val rationalLess : RATIONAL → RATIONAL → BOOL
val rationalLessEqual : RATIONAL → RATIONAL → BOOL
val rationalGreater : RATIONAL → RATIONAL → BOOL
val rationalGreaterEqual : RATIONAL → RATIONAL → BOOL

declare hol target_rep function rationalLess = infix '<'
declare ocaml target_rep function rationalLess = 'Rational.lt'
declare isabelle target_rep function rationalLess = infix '<'
declare coq target_rep function rationalLess = 'Qlt_bool'

declare hol target_rep function rationalLessEqual = infix '<='
declare ocaml target_rep function rationalLessEqual = 'Rational.leq'
declare isabelle target_rep function rationalLessEqual = infix '\<le>'
declare coq target_rep function rationalLessEqual = 'Qle_bool'

declare hol target_rep function rationalGreater = infix '>'
declare ocaml target_rep function rationalGreater = 'Rational.gt'
declare isabelle target_rep function rationalGreater = infix '>'
declare coq target_rep function rationalGreater = 'Qgt_bool'

declare hol target_rep function rationalGreaterEqual = infix '>='
declare ocaml target_rep function rationalGreaterEqual = 'Rational.geq'
declare isabelle target_rep function rationalGreaterEqual = infix '\<ge>'
```

```
declare coq target_rep function rationalGreaterEqual = 'Qge_bool'

val rationalCompare : RATIONAL → RATIONAL → ORDERING
let inline rationalCompare = defaultCompare
let inline {coq, isabelle, hol, ocaml} rationalCompare = genericCompare rationalLess rationalEq

instance (Ord RATIONAL)
  let compare = rationalCompare
  let < = rationalLess
  let <= = rationalLessEqual
  let > = rationalGreater
  let >= = rationalGreaterEqual
end

instance (SetType RATIONAL)
  let setElemCompare = rationalCompare
end

val rationalAdd : RATIONAL → RATIONAL → RATIONAL
declare hol target_rep function rationalAdd = infix '+'
declare ocaml target_rep function rationalAdd = 'Rational.add'
declare isabelle target_rep function rationalAdd = infix '+'
declare coq target_rep function rationalAdd = 'Qplus'

instance (NumAdd RATIONAL)
  let + = rationalAdd
end

val rationalMinus : RATIONAL → RATIONAL → RATIONAL
declare hol target_rep function rationalMinus = infix '-'
declare ocaml target_rep function rationalMinus = 'Rational.sub'
declare isabelle target_rep function rationalMinus = infix '-'
declare coq target_rep function rationalMinus = 'Qminus'

instance (NumMinus RATIONAL)
  let - = rationalMinus
end

val rationalNegate : RATIONAL → RATIONAL
let inline rationalNegate n = 0 - n
declare ocaml target_rep function rationalNegate = 'Rational.neg'
declare isabelle target_rep function rationalNegate i = '-' i

instance (NumNegate RATIONAL)
  let ~ = rationalNegate
end

val rationalAbs : RATIONAL → RATIONAL
let inline rationalAbs n = (if n > 0 then n else -n)
declare ocaml target_rep function rationalAbs = 'Rational.abs'
declare isabelle target_rep function rationalAbs = 'abs'

instance (NumAbs RATIONAL)
  let abs = rationalAbs
end

val rationalSucc : RATIONAL → RATIONAL
let inline rationalSucc n = n + 1
```

```
instance (NumSucc RATIONAL)
  let succ =  rationalSucc
end

val rationalPred : RATIONAL → RATIONAL
let inline rationalPred n = n − 1
instance (NumPred RATIONAL)
  let pred =  rationalPred
end

val rationalMult : RATIONAL → RATIONAL → RATIONAL
declare hol target_rep function rationalMult = infix '*'
declare ocaml target_rep function rationalMult = 'Rational.mul'
declare isabelle target_rep function rationalMult = infix '*'
declare coq target_rep function rationalMult = 'Qmult'

instance (NumMult RATIONAL)
  let ∗ =  rationalMult
end

val rationalDiv : RATIONAL → RATIONAL → RATIONAL
declare hol target_rep function rationalDiv = infix '/'
declare ocaml target_rep function rationalDiv = 'Rational.div'
declare isabelle target_rep function rationalDiv = infix 'div'
declare coq target_rep function rationalDiv = 'Qdiv'

instance ( NumDivision RATIONAL )
  let / =  rationalDiv
end

val rationalFromFrac : INT → INT → RATIONAL
let rationalFromFrac n d =  (rationalFromInt n) / (rationalFromInt d)
declare ocaml target_rep function rationalFromFrac n d = ('Rational.of_ints' n d)
declare isabelle target_rep function rationalFromFrac n d = ('Fract' n d)
declare hol target_rep function rationalFromFrac n d = ('rat_cons' n d)

val rationalNumerator : RATIONAL → $\MATHBB{Z}$
declare ocaml target_rep function rationalNumerator r = ('Rational.num' r)
declare isabelle target_rep function rationalNumerator r = ('fst' ('quotient_of' r))
declare hol target_rep function rationalNumerator r = ('Numerator' r)
declare coq target_rep function rationalNumerator r = ('Qnum' r) (∗ TODO : test ∗)

val rationalDenominator : RATIONAL → $\MATHBB{Z}$
declare ocaml target_rep function rationalDenominator r = ('Rational.den' r)
declare isabelle target_rep function rationalDenominator r = ('snd' ('quotient_of' r))
declare hol target_rep function rationalDenominator r = ('Denominator' r)
declare coq target_rep function rationalDenominator r = ('QDen' r) (∗ TODO : test ∗)

val rationalPowInteger : RATIONAL → $\MATHBB{Z}$ → RATIONAL
let rec rationalPowInteger b e =
   if e = 0 then 1 else
  if e > 0 then rationalPowInteger b (e − 1) ∗ b else
  rationalPowInteger b (e + 1) / b
declare coq target_rep function rationalPowInteger = 'Qpower'
declare {isabelle} termination_argument rationalPowInteger = automatic

val rationalPowNat : RATIONAL → NAT → RATIONAL
let rationalPowNat r e =  rationalPowInteger r (integerFromNat e)
```

```
declare isabelle target_rep function rationalPowNat = 'power'
declare coq target_rep function rationalPowNat r e = ('Qpower' r ('Z.of_nat' e))

instance ( NumPow RATIONAL )
  let ** =  rationalPowNat
end

val rationalMin : RATIONAL → RATIONAL → RATIONAL
let inline rationalMin = defaultMin
declare isabelle target_rep function rationalMin = 'min'
declare ocaml target_rep function rationalMin = 'Rational.min'
declare coq target_rep function rationalMin = 'Qmin'

val rationalMax : RATIONAL → RATIONAL → RATIONAL
let inline rationalMax = defaultMax
declare isabelle target_rep function rationalMax = 'max'
declare ocaml target_rep function rationalMax = 'Rational.max'
declare coq target_rep function rationalMax = 'Qmax'

instance ( OrdMaxMin RATIONAL )
  let max =  rationalMax
  let min =  rationalMin
end



(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* real                                                   *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val realFromNumeral : NUMERAL → REAL
declare ocaml target_rep function realFromNumeral n = ('Nat_big_num.to_float' n)
declare isabelle target_rep function realFromNumeral n = (''n : REAL)
declare hol target_rep function realFromNumeral n = ('real_of_num' n)
declare coq target_rep function realFromNumeral n = ('IZR' ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))))

instance (Numeral REAL)
  let fromNumeral n =  realFromNumeral n
end

val realFromInteger : $\MATHBB{Z}$ → REAL
declare ocaml target_rep function realFromInteger n = ('float_of_int' ('Nat_big_num.to_int' n))
declare isabelle target_rep function realFromInteger n = ('real_of_int' n)
declare hol target_rep function realFromInteger n = ('real_of_int' n)
declare coq target_rep function realFromInteger n = ('IZR' n)

val realEq : REAL → REAL → BOOL
let inline realEq = unsafe_structural_equality
declare coq target_rep function realEq = 'Reqb'
instance (Eq REAL)
  let = =  realEq
  let <> n1 n2 =  ¬ (realEq n1 n2)
end

val realLess : REAL → REAL → BOOL
val realLessEqual : REAL → REAL → BOOL
val realGreater : REAL → REAL → BOOL
val realGreaterEqual : REAL → REAL → BOOL
```

```
declare hol target_rep function realLess = infix '<'
declare ocaml target_rep function realLess = infix '<'
declare isabelle target_rep function realLess = infix '<'
declare coq target_rep function realLess = 'Rlt_bool'

declare hol target_rep function realLessEqual = infix '<='
declare ocaml target_rep function realLessEqual = infix '<='
declare isabelle target_rep function realLessEqual = infix '\<le>'
declare coq target_rep function realLessEqual = 'Rle_bool'

declare hol target_rep function realGreater = infix '>'
declare ocaml target_rep function realGreater = infix '>'
declare isabelle target_rep function realGreater = infix '>'
declare coq target_rep function realGreater = 'Rgt_bool'

declare hol target_rep function realGreaterEqual = infix '>='
declare ocaml target_rep function realGreaterEqual = infix '>='
declare isabelle target_rep function realGreaterEqual = infix '\<ge>'
declare coq target_rep function realGreaterEqual = 'Rge_bool'

val realCompare : REAL → REAL → ORDERING
let inline realCompare = defaultCompare
let inline {coq, isabelle, hol, ocaml} realCompare = genericCompare realLess realEq

instance (Ord REAL)
  let compare = realCompare
  let < = realLess
  let <= = realLessEqual
  let > = realGreater
  let >= = realGreaterEqual
end

instance (SetType REAL)
  let setElemCompare = realCompare
end

val realAdd : REAL → REAL → REAL
declare hol target_rep function realAdd = infix '+'
declare ocaml target_rep function realAdd = 'Lem.plus_float'
declare isabelle target_rep function realAdd = infix '+'
declare coq target_rep function realAdd = 'Rplus'

instance (NumAdd REAL)
  let + = realAdd
end

val realMinus : REAL → REAL → REAL
declare hol target_rep function realMinus = infix '-'
declare ocaml target_rep function realMinus = 'Lem.minus_float'
declare isabelle target_rep function realMinus = infix '-'
declare coq target_rep function realMinus = 'Rminus'

instance (NumMinus REAL)
  let − = realMinus
end

val realNegate : REAL → REAL
```

```
let inline realNegate n = 0 − n
declare ocaml target_rep function realNegate = 'Lem.neg_float'
declare isabelle target_rep function realNegate i = '-' i
declare coq target_rep function realNegate = 'Ropp'

instance (NumNegate REAL)
  let ~ = realNegate
end

val realAbs : REAL → REAL
let inline realAbs n = (if n > 0 then n else −n)
declare ocaml target_rep function realAbs = 'abs_float'
declare isabelle target_rep function realAbs = 'abs'
declare coq target_rep function realAbs = 'Rabs'

instance (NumAbs REAL)
  let abs = realAbs
end

val realSucc : REAL → REAL
let inline realSucc n = n + 1
instance (NumSucc REAL)
  let succ = realSucc
end

val realPred : REAL → REAL
let inline realPred n = n − 1
instance (NumPred REAL)
  let pred = realPred
end

val realMult : REAL → REAL → REAL
declare hol target_rep function realMult = infix '*'
declare ocaml target_rep function realMult = 'Lem.mult_float'
declare isabelle target_rep function realMult = infix '*'
declare coq target_rep function realMult = 'Rmult'

instance (NumMult REAL)
  let * = realMult
end

val realDiv : REAL → REAL → REAL
declare hol target_rep function realDiv = infix '/'
declare ocaml target_rep function realDiv = 'Lem.div_float'
declare isabelle target_rep function realDiv = infix 'div'
declare coq target_rep function realDiv = 'Rdiv'

instance ( NumDivision REAL )
  let / = realDiv
end

val realFromFrac : $\MATHBB{Z}$ → $\MATHBB{Z}$ → REAL
let realFromFrac n d = realDiv (realFromInteger n) (realFromInteger d)
declare ocaml target_rep function realFromFrac n d = ('Lem.div_float' (realFromInteger n) (realFromInteger d))


val realPowInteger : REAL → $\MATHBB{Z}$ → REAL
let rec realPowInteger b e =
```

```
  if e = 0 then 1 else
 if e > 0 then realPowInteger b (e − 1) ∗ b else
 realPowInteger b (e + 1) / b
declare ocaml target_rep function realPowInteger r e = ('Lem.pow_float' r (realFromInteger e))
declare coq target_rep function realPowInteger = 'powerRZ'
declare {isabelle} termination_argument realPowInteger = automatic

val realPowNat : REAL → NAT → REAL
let realPowNat r e = realPowInteger r (integerFromNat e)
declare isabelle target_rep function realPowNat = 'power'
declare coq target_rep function realPowNat = 'pow'
declare hol target_rep function realPowNat = infix 'pow'

instance ( NumPow REAL )
 let ∗∗ = realPowNat
end

val realSqrt : REAL → REAL
declare hol target_rep function realSqrt = 'sqrt'
declare ocaml target_rep function realSqrt = 'sqrt'
declare isabelle target_rep function realSqrt = 'sqrt'
declare coq target_rep function realSqrt = 'Rsqrt'

val realMin : REAL → REAL → REAL
let inline realMin = defaultMin
declare hol target_rep function realMin = 'min'
declare isabelle target_rep function realMin = 'min'
declare ocaml target_rep function realMin = 'min'
declare coq target_rep function realMin = 'Rmin'

val realMax : REAL → REAL → REAL
let inline realMax = defaultMax
declare hol target_rep function realMax = 'max'
declare isabelle target_rep function realMax = 'max'
declare ocaml target_rep function realMax = 'max'
declare coq target_rep function realMax = 'Rmax'

instance ( OrdMaxMin REAL )
 let max = realMax
 let min = realMin
end

val realCeiling : REAL → $\MATHBB{Z}$
declare isabelle target_rep function realCeiling = 'ceiling'
declare ocaml target_rep function realCeiling = 'Lem.big_num_of_ceil'
declare hol target_rep function realCeiling = 'clg'
declare coq target_rep function realCeiling = 'up'

val realFloor : REAL → $\MATHBB{Z}$
declare isabelle target_rep function realFloor = 'floor'
declare ocaml target_rep function realFloor = 'Lem.big_num_of_floor'
declare hol target_rep function realFloor = 'flr'
declare coq target_rep function realFloor = 'Rdown'


val integerSqrt : $\MATHBB{Z}$ → $\MATHBB{Z}$
let integerSqrt i = realFloor (realSqrt (realFromInteger i))
declare ocaml target_rep function integerSqrt = 'Nat_big_num.sqrt'
```

```
declare coq target_rep function integerSqrt = 'Z.sqrt'


(* ========================================================================== *)
(* Tests                                                                      *)
(* ========================================================================== *)

assert nat_test1 : (2 + (5 : NAT) = 7)
assert nat_test2 : (8 - (7 : NAT) = 1)
assert nat_test3 : (7 - (8 : NAT) = 0)
assert nat_test4 : (7 * (8 : NAT) = 56)
assert nat_test5 : ((7 : NAT)^2 = 49)
assert nat_test6 : (div 11 (4 : NAT) = 2)
assert nat_test7 : (11 / (4 : NAT) = 2)
assert nat_test8 : (11 mod (4 : NAT) = 3)
assert nat_test9 : (11 < (12 : NAT))
assert nat_test10 : (11 ≤ (12 : NAT))
assert nat_test11 : (12 ≤ (12 : NAT))
assert nat_test12 : (¬ (12 < (12 : NAT)))
assert nat_test13 : (12 > (11 : NAT))
assert nat_test14 : (12 ≥ (11 : NAT))
assert nat_test15 : (12 ≥ (12 : NAT))
assert nat_test16 : (¬ (12 > (12 : NAT)))
assert nat_test17 : (min 12 (12 : NAT) = 12)
assert nat_test18 : (min 10 (12 : NAT) = 10)
assert nat_test19 : (min 12 (10 : NAT) = 10)
assert nat_test20 : (max 12 (12 : NAT) = 12)
assert nat_test21 : (max 10 (12 : NAT) = 12)
assert nat_test22 : (max 12 (10 : NAT) = 12)
assert nat_test23 : (succ 12 = (13 : NAT))
assert nat_test24 : (succ 0 = (1 : NAT))
assert nat_test25 : (pred 12 = (11 : NAT))
assert nat_test26 : (pred 0 = (0 : NAT))
assert nat_test27 : (match (27 : NAT) with
  | 0 → false
  | x + 2 → (x = 25)
  | x + 1 → (x = 26)
 end)
assert nat_test28a : (match (27 : NAT) with
  | n + 50 → "50 <= x"
  | 40 → "x = 40"
  | n + 31 → "x <> 40 && 31 <= x < 50"
  | 29 → "x = 29"
  | n + 30 → "x = 30"
  | 4 → "x = 4"
  | _ → "x <> 4 && x <> 29 && x < 30"
 end = "x <> 4 && x <> 29 && x < 30")
assert nat_test28b : (match (30 : NAT) with
  | n + 50 → "50 <= x"
  | 40 → "x = 40"
  | n + 31 → "x <> 40 && 31 <= x < 50"
  | 29 → "x = 29"
  | n + 30 → "x = 30"
  | 4 → "x = 4"
  | _ → "x <> 4 && x <> 29 && x < 30"
 end = "x = 30")
                (*assert nat_test29 : (0x7F + (0x01 : nat) = 0x80)*)
```

```
assert natural_test1 : (2 + (5 : $\mathbb{N}$) = 7)
assert natural_test2 : (8 − (7 : $\mathbb{N}$) = 1)
assert natural_test3 : (7 − (8 : $\mathbb{N}$) = 0)
assert natural_test4 : (7 ∗ (8 : $\mathbb{N}$) = 56)
assert natural_test5 : ((7 : $\mathbb{N}$)^2 = 49)
assert natural_test6 : (div 11 (4 : $\mathbb{N}$) = 2)
assert natural_test7 : (11 / (4 : $\mathbb{N}$) = 2)
assert natural_test8 : (11 mod (4 : $\mathbb{N}$) = 3)
assert natural_test9 : (11 < (12 : $\mathbb{N}$))
assert natural_test10 : (11 ≤ (12 : $\mathbb{N}$))
assert natural_test11 : (12 ≤ (12 : $\mathbb{N}$))
assert natural_test12 : (¬ (12 < (12 : $\mathbb{N}$)))
assert natural_test13 : (12 > (11 : $\mathbb{N}$))
assert natural_test14 : (12 ≥ (11 : $\mathbb{N}$))
assert natural_test15 : (12 ≥ (12 : $\mathbb{N}$))
assert natural_test16 : (¬ (12 > (12 : $\mathbb{N}$)))
assert natural_test17 : (min 12 (12 : $\mathbb{N}$) = 12)
assert natural_test18 : (min 10 (12 : $\mathbb{N}$) = 10)
assert natural_test19 : (min 12 (10 : $\mathbb{N}$) = 10)
assert natural_test20 : (max 12 (12 : $\mathbb{N}$) = 12)
assert natural_test21 : (max 10 (12 : $\mathbb{N}$) = 12)
assert natural_test22 : (max 12 (10 : $\mathbb{N}$) = 12)
assert natural_test23 : (succ 12 = (13 : $\mathbb{N}$))
assert natural_test24 : (succ 0 = (1 : $\mathbb{N}$))
assert natural_test25 : (pred 12 = (11 : $\mathbb{N}$))
assert natural_test26 : (pred 0 = (0 : $\mathbb{N}$))
assert natural_test27 : (match (27 : $\mathbb{N}$) with
  | 0 → false
  | x + 2 → (x = 25)
  | x + 1 → (x = 26)
 end)
assert natural_test28a : (match (27 : $\mathbb{N}$) with
  | n + 50 → "50 <= x"
  | 40 → "x = 40"
  | n + 31 → "x <> 40 && 31 <= x < 50"
  | 29 → "x = 29"
  | n + 30 → "x = 30"
  | 4 → "x = 4"
  | _ → "x <> 4 && x <> 29 && x < 30"
 end = "x <> 4 && x <> 29 && x < 30")
assert natural_test28b : (match (30 : $\mathbb{N}$) with
  | n + 50 → "50 <= x"
  | 40 → "x = 40"
  | n + 31 → "x <> 40 && 31 <= x < 50"
  | 29 → "x = 29"
  | n + 30 → "x = 30"
  | 4 → "x = 4"
  | _ → "x <> 4 && x <> 29 && x < 30"
 end = "x = 30")
                    (*assert natural_test29 : (0x7F + (0x01 : natural) = 0x80)*)


assert int_test1 : (2 + (5 : int) = 7)
assert int_test2 : (8 − (7 : int) = 1)
assert int_test3 : (7 − (8 : int) = −1)
assert int_test4 : (7 ∗ (8 : int) = 56)
```

assert int_test_5 : $((7 : \text{INT})^2 = 49)$
assert int_test_6 : $(\text{div } 11\ (4 : \text{INT}) = 2)$
assert int_test6a : $(\text{div } (-11)\ (4 : \text{INT}) = -3)$
assert int_test_7 : $(11\ /\ (4 : \text{INT}) = 2)$
assert int_test7a : $(-11\ /\ (4 : \text{INT}) = -3)$
assert int_test_8 : $(11 \bmod (4 : \text{INT}) = 3)$
assert $int\_test8at$: $(-11 \bmod (4 : \text{INT}) = 1)$
assert int_test_9 : $(11 < (12 : \text{INT}))$
assert int_test_{10} : $(11 \leq (12 : \text{INT}))$
assert int_test_{11} : $(12 \leq (12 : \text{INT}))$
assert int_test_{12} : $(\neg\ (12 < (12 : \text{INT})))$
assert int_test_{13} : $(12 > (11 : \text{INT}))$
assert int_test_{14} : $(12 \geq (11 : \text{INT}))$
assert int_test_{15} : $(12 \geq (12 : \text{INT}))$
assert int_test_{16} : $(\neg\ (12 > (12 : \text{INT})))$
assert int_test_{17} : $(\min\ 12\ (12 : \text{INT}) = 12)$
assert int_test_{18} : $(\min\ 10\ (12 : \text{INT}) = 10)$
assert int_test_{19} : $(\min\ 12\ (10 : \text{INT}) = 10)$
assert int_test_{20} : $(\max\ 12\ (12 : \text{INT}) = 12)$
assert int_test_{21} : $(\max\ 10\ (12 : \text{INT}) = 12)$
assert int_test_{22} : $(\max\ 12\ (10 : \text{INT}) = 12)$
assert int_test_{23} : $(\text{succ } 12 = (13 : \text{INT}))$
assert int_test_{24} : $(\text{succ } 0 = (1 : \text{INT}))$
assert int_test_{25} : $(\text{pred } 12 = (11 : \text{INT}))$
assert int_test_{26} : $(\text{pred } 0 = -(1 : \text{INT}))$
assert int_test_{27} : $(\text{abs } 42 = (42 : \text{INT}))$
assert int_test_{28} : $(\text{abs } (-42) = (42 : \text{INT}))$
(*assert int_test29 : (0x7F + (0x01 : int) = 0x80)*)

assert $int32\_test_1$: $(2 + (5 : \text{INT}_{32}) = 7)$
assert $int32\_test_2$: $(8 - (7 : \text{INT}_{32}) = 1)$
assert $int32\_test_3$: $(7 - (8 : \text{INT}_{32}) = -1)$
assert $int32\_test_4$: $(7 * (8 : \text{INT}_{32}) = 56)$
assert $int32\_test_5$: $((7 : \text{INT}_{32})^2 = 49)$
assert $int32\_test_6$: $(\text{div } 11\ (4 : \text{INT}_{32}) = 2)$
assert $int32\_test_7$: $(11\ /\ (4 : \text{INT}_{32}) = 2)$
assert $int32\_test_8$: $(11 \bmod (4 : \text{INT}_{32}) = 3)$
assert $int32\_test_9$: $(11 < (12 : \text{INT}_{32}))$
assert $int32\_test_{10}$: $(11 \leq (12 : \text{INT}_{32}))$
assert $int32\_test_{11}$: $(12 \leq (12 : \text{INT}_{32}))$
assert $int32\_test_{12}$: $(\neg\ (12 < (12 : \text{INT}_{32})))$
assert $int32\_test_{13}$: $(12 > (11 : \text{INT}_{32}))$
assert $int32\_test13a$: $(12 > (-(11 : \text{INT}_{32})))$
assert $int32\_test_{14}$: $(12 \geq (11 : \text{INT}_{32}))$
assert $int32\_test_{15}$: $(12 \geq (12 : \text{INT}_{32}))$
assert $int32\_test_{16}$: $(\neg\ (12 > (12 : \text{INT}_{32})))$
assert $int32\_test_{17}$: $(\min\ 12\ (12 : \text{INT}_{32}) = 12)$
assert $int32\_test_{18}$: $(\min\ 10\ (12 : \text{INT}_{32}) = 10)$
assert $int32\_test_{19}$: $(\min\ 12\ (10 : \text{INT}_{32}) = 10)$
assert $int32\_test_{20}$: $(\max\ 12\ (12 : \text{INT}_{32}) = 12)$
assert $int32\_test_{21}$: $(\max\ (-10)\ (12 : \text{INT}_{32}) = 12)$
assert $int32\_test_{22}$: $(\max\ 12\ (10 : \text{INT}_{32}) = 12)$
assert $int32\_test_{23}$: $(\text{succ } 12 = (13 : \text{INT}_{32}))$
assert $int32\_test_{24}$: $(\text{succ } 0 = (1 : \text{INT}_{32}))$
assert $int32\_test_{25}$: $(\text{pred } 12 = (11 : \text{INT}_{32}))$
assert $int32\_test_{26}$: $(\text{pred } 0 = -(1 : \text{INT}_{32}))$
assert $int32\_test_{27}$: $(\text{abs } 42 = (42 : \text{INT}_{32}))$
assert $int32\_test_{28}$: $(\text{abs } (-42) = (42 : \text{INT}_{32}))$

assert $int64\_test_1$: $(2 + (5 : \text{INT}_{64}) = 7)$
assert $int64\_test_2$: $(8 - (7 : \text{INT}_{64}) = 1)$
assert $int64\_test_3$: $(7 - (8 : \text{INT}_{64}) = -1)$
assert $int64\_test_4$: $(7 * (8 : \text{INT}_{64}) = 56)$
assert $int64\_test_5$: $((7 : \text{INT}_{64})^2 = 49)$
assert $int64\_test_6$: $(\text{div } 11\ (4 : \text{INT}_{64}) = 2)$
assert $int64\_test_7$: $(11 / (4 : \text{INT}_{64}) = 2)$
assert $int64\_test_8$: $(11 \bmod (4 : \text{INT}_{64}) = 3)$
assert $int64\_test_9$: $(11 < (12 : \text{INT}_{64}))$
assert $int64\_test_{10}$: $(11 \leq (12 : \text{INT}_{64}))$
assert $int64\_test_{11}$: $(12 \leq (12 : \text{INT}_{64}))$
assert $int64\_test_{12}$: $(\neg (12 < (12 : \text{INT}_{64})))$
assert $int64\_test_{13}$: $(12 > (11 : \text{INT}_{64}))$
assert $int64\_test13a$: $(12 > (-(11 : \text{INT}_{64})))$
assert $int64\_test_{14}$: $(12 \geq (11 : \text{INT}_{64}))$
assert $int64\_test_{15}$: $(12 \geq (12 : \text{INT}_{64}))$
assert $int64\_test_{16}$: $(\neg (12 > (12 : \text{INT}_{64})))$
assert $int64\_test_{17}$: $(\min\ 12\ (12 : \text{INT}_{64}) = 12)$
assert $int64\_test_{18}$: $(\min\ 10\ (12 : \text{INT}_{64}) = 10)$
assert $int64\_test_{19}$: $(\min\ 12\ (10 : \text{INT}_{64}) = 10)$
assert $int64\_test_{20}$: $(\max\ 12\ (12 : \text{INT}_{64}) = 12)$
assert $int64\_test_{21}$: $(\max\ (-10)\ (12 : \text{INT}_{64}) = 12)$
assert $int64\_test_{22}$: $(\max\ 12\ (10 : \text{INT}_{64}) = 12)$
assert $int64\_test_{23}$: $(\text{succ } 12 = (13 : \text{INT}_{64}))$
assert $int64\_test_{24}$: $(\text{succ } 0 = (1 : \text{INT}_{64}))$
assert $int64\_test_{25}$: $(\text{pred } 12 = (11 : \text{INT}_{64}))$
assert $int64\_test_{26}$: $(\text{pred } 0 = -(1 : \text{INT}_{64}))$
assert $int64\_test_{27}$: $(\text{abs } 42 = (42 : \text{INT}_{64}))$
assert $int64\_test_{28}$: $(\text{abs } (-42) = (42 : \text{INT}_{64}))$

assert $integer\_test_1$: $(2 + (5 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 7)$
assert $integer\_test_2$: $(8 - (7 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 1)$
assert $integer\_test_3$: $(7 - (8 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = -1)$
assert $integer\_test_4$: $(7 * (8 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 56)$
assert $integer\_test_5$: $((7 : \$\backslash\text{MATHBB}\{\text{Z}\}\$)^2 = 49)$
assert $integer\_test_6$: $(\text{div } 11\ (4 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 2)$
assert $integer\_test6a$: $(\text{div } (-11)\ (4 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = -3)$
assert $integer\_test_7$: $(11 / (4 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 2)$
assert $integer\_test7a$: $(-11 / (4 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = -3)$
assert $integer\_test_8$: $(11 \bmod (4 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 3)$
assert $integer\_test8a$: $(-11 \bmod (4 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 1)$
assert $integer\_test_9$: $(11 < (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$))$
assert $integer\_test_{10}$: $(11 \leq (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$))$
assert $integer\_test_{11}$: $(12 \leq (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$))$
assert $integer\_test_{12}$: $(\neg (12 < (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$)))$
assert $integer\_test_{13}$: $(12 > (11 : \$\backslash\text{MATHBB}\{\text{Z}\}\$))$
assert $integer\_test_{14}$: $(12 \geq (11 : \$\backslash\text{MATHBB}\{\text{Z}\}\$))$
assert $integer\_test_{15}$: $(12 \geq (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$))$
assert $integer\_test_{16}$: $(\neg (12 > (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$)))$
assert $integer\_test_{17}$: $(\min\ 12\ (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 12)$
assert $integer\_test_{18}$: $(\min\ 10\ (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 10)$
assert $integer\_test_{19}$: $(\min\ 12\ (10 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 10)$
assert $integer\_test_{20}$: $(\max\ 12\ (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 12)$
assert $integer\_test_{21}$: $(\max\ 10\ (12 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 12)$
assert $integer\_test_{22}$: $(\max\ 12\ (10 : \$\backslash\text{MATHBB}\{\text{Z}\}\$) = 12)$
assert $integer\_test_{23}$: $(\text{succ } 12 = (13 : \$\backslash\text{MATHBB}\{\text{Z}\}\$))$
assert $integer\_test_{24}$: $(\text{succ } 0 = (1 : \$\backslash\text{MATHBB}\{\text{Z}\}\$))$

```
assert integer_test25 : (pred 12 = (11 : $\mathbb{Z}$))
assert integer_test26 : (pred 0 = -(1 : $\mathbb{Z}$))
assert integer_test27 : (abs 42 = (42 : $\mathbb{Z}$))
assert integer_test28 : (abs (-42) = (42 : $\mathbb{Z}$))
assert integer_test29 : (integerSqrt 5 = 2)
            (*assert integer_test30 : (0xFFFFFFFFFFFFFFFF + (0b1 : integer) = 0x10000000000000000)*)


assert rational_test1 : (2 + (5 : rational) = 7)
assert rational_test2 : ((rationalFromFrac 3 2) + (rationalFromFrac 1 2) = 2)
assert rational_test3 : (7 - (8 : rational) = -1)
assert rational_test4 : (7 * (8 : rational) = 56)
assert rational_test5 : ((7 : rational)^2 = 49)
assert rational_test5a : (rationalPowInteger (2 : rational) (-3) = rationalFromFrac 1 8)
assert rational_test5b : (rationalPowInteger (-2 : rational) (-3) = rationalFromFrac (-1) 8)
assert rational_test5c : (rationalPowInteger (-2 : rational) (-2) = rationalFromFrac 1 4)
assert rational_test6 : (11 / (4 : rational) = (rationalFromFrac 11 4))
assert rational_test6a : ((- 11) / (4 : rational) = (rationalFromFrac (-11) 4))
assert rational_test7 : (11 < (12 : rational))
assert rational_test8 : (11 ≤ (12 : rational))
assert rational_test9 : (12 ≤ (12 : rational))
assert rational_test10 : (¬ (12 < (12 : rational)))
assert rational_test11 : (12 > (11 : rational))
assert rational_test12 : (12 ≥ (11 : rational))
assert rational_test13 : (12 ≥ (12 : rational))
assert rational_test14 : (¬ (12 > (12 : rational)))
assert rational_test15 : (min 12 (12 : rational) = 12)
assert rational_test16 : (min 10 (12 : rational) = 10)
assert rational_test17 : (min 12 (10 : rational) = 10)
assert rational_test18 : (max 12 (12 : rational) = 12)
assert rational_test19 : (max 10 (12 : rational) = 12)
assert rational_test20 : (max 12 (10 : rational) = 12)
assert rational_test21 : (succ 12 = (13 : rational))
assert rational_test22 : (succ 0 = (1 : rational))
assert rational_test23 : (pred 12 = (11 : rational))
assert rational_test24 : (pred 0 = -(1 : rational))
assert rational_test25 : (abs 42 = (42 : rational))
assert rational_test26 : (abs (-42) = (42 : rational))
assert rational_test27 : ((rationalFromFrac 1 2) * 2 = 1)
assert rational_test28 :
  (let r = rationalFromFrac (-11) 4 in
   (rationalFromInteger (rationalNumerator r) / rationalFromInteger (rationalDenominator r) = r))
assert rational_test29 :
  (let r = rationalFromFrac 8 4 in
  (rationalFromInteger (rationalNumerator r) / rationalFromInteger (rationalDenominator r) = rationalFromInt 2))


assert real_test1 : (2 + (5 : real) = 7)
assert real_test2 : ((3 / (2 : real)) + (1 / 2) = 2)
assert real_test3 : (7 - (8 : real) = -1)
assert real_test4 : (7 * (8 : real) = 56)
assert real_test5 : ((7 : real)^2 = 49)
assert real_test5a : (realPowInteger (2 : real) (-3) = realFromFrac 1 8)
assert real_test5b : (realPowInteger (-2 : real) (-3) = realFromFrac (-1) 8)
assert real_test5c : (realPowInteger (-2 : real) (-2) = realFromFrac 1 4)
assert real_test6 : (11 / (4 : real) = (realFromFrac 11 4))
assert real_test6a : ((- 11) / (4 : real) = (realFromFrac (-11) 4))
assert real_test7 : (11 < (12 : real))
```

assert $real\_test_8$: $(11 \leq (12 : \text{REAL}))$
assert $real\_test_9$: $(12 \leq (12 : \text{REAL}))$
assert $real\_test_{10}$: $(\neg\ (12 < (12 : \text{REAL})))$
assert $real\_test_{11}$: $(12 > (11 : \text{REAL}))$
assert $real\_test_{12}$: $(12 \geq (11 : \text{REAL}))$
assert $real\_test_{13}$: $(12 \geq (12 : \text{REAL}))$
assert $real\_test_{14}$: $(\neg\ (12 > (12 : \text{REAL})))$
assert $real\_test_{15}$: (min 12 (12 : REAL) = 12)
assert $real\_test_{16}$: (min 10 (12 : REAL) = 10)
assert $real\_test_{17}$: (min 12 (10 : REAL) = 10)
assert $real\_test_{18}$: (max 12 (12 : REAL) = 12)
assert $real\_test_{19}$: (max 10 (12 : REAL) = 12)
assert $real\_test_{20}$: (max 12 (10 : REAL) = 12)
assert $real\_test_{21}$: (succ 12 = (13 : REAL))
assert $real\_test_{22}$: (succ 0 = (1 : REAL))
assert $real\_test_{23}$: (pred 12 = (11 : REAL))
assert $real\_test_{24}$: (pred 0 = $-$(1 : REAL))
assert $real\_test_{25}$: (abs 42 = (42 : REAL))
assert $real\_test_{26}$: (abs (-42) = (42 : REAL))
assert $real\_test_{27}$: ((1 / (2 : REAL)) $*$ 2 = 1)
assert $real\_test_{28}$: (realFloor (realFromFrac 11 4) = 2)
assert $real\_test_{29}$: (realCeiling (realFromFrac 11 4) = 3)
assert $real\_test_{30}$: (realFloor (realFromFrac 12 4) = 3)
assert $real\_test_{31}$: (realCeiling (realFromFrac 12 4) = 3)
assert $real\_test_{32}$: (realFloor (realFromFrac (-3) 2) = -2)
assert $real\_test_{33}$: (realCeiling (realFromFrac (-3) 2) = -1)

(* ========== *)
(* Translation between number types *)
(* ========== *)

(**********)
(* integerFrom... *)
(**********)

val *integerFromInt* : INT → $\MATHBB{Z}$
declare *hol* target_rep function integerFromInt = '' (* remove natFromNumeral, as it is the identify function *)
declare *ocaml* target_rep function integerFromInt = 'Nat_big_num.of_int'
declare *isabelle* target_rep function integerFromInt = ''
declare *coq* target_rep function integerFromInt = ''

assert $integer\_from\_int_0$: integerFromInt 0 = 0
assert $integer\_from\_int_1$: integerFromInt 1 = 1
assert $integer\_from\_int_2$: integerFromInt (-2) = (-2)

assert $integer\_from\_nat_0$: integerFromNat 0 = 0
assert $integer\_from\_nat_1$: integerFromNat 1 = 1
assert $integer\_from\_nat_2$: integerFromNat 12 = 12

val *integerFromNatural* : $\MATHBB{N}$ → $\MATHBB{Z}$
declare *hol* target_rep function integerFromNatural = 'int_of_num'
declare *ocaml* target_rep function integerFromNatural n = ''n
declare *isabelle* target_rep function integerFromNatural = 'int'
declare *coq* target_rep function integerFromNatural n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))) (* TODO : check *)

assert $integerFromNatural_0$: integerFromNatural 0 = 0
assert $integerFromNatural_1$: integerFromNatural 822 = 822
assert $integerFromNatural_2$: integerFromNatural 12 = 12

val *integerFromInt*$_{32}$: INT$_{32}$ → $\MATHBB{Z}$
declare *ocaml* target_rep function integerFromInt$_{32}$ = 'Nat_big_num.of_int'$_{32}$
declare *isabelle* target_rep function integerFromInt$_{32}$ = 'sint'
declare *hol* target_rep function integerFromInt$_{32}$ = 'w2int'
declare *coq* target_rep function integerFromInt$_{32}$ = ''

assert *integer_from_int*$_{32}$_0 : integerFromInt$_{32}$ 0 = 0
assert *integer_from_int*$_{32}$_1 : integerFromInt$_{32}$ 1 = 1
assert *integer_from_int*$_{32}$_2 : integerFromInt$_{32}$ 123 = 123
assert *integer_from_int*$_{32}$_3 : integerFromInt$_{32}$ (−0) = −0
assert *integer_from_int*$_{32}$_4 : integerFromInt$_{32}$ (−1) = −1
assert *integer_from_int*$_{32}$_5 : integerFromInt$_{32}$ (−123) = −123

val *integerFromInt*$_{64}$: INT$_{64}$ → $\MATHBB{Z}$
declare *ocaml* target_rep function integerFromInt$_{64}$ = 'Nat_big_num.of_int'$_{64}$
declare *isabelle* target_rep function integerFromInt$_{64}$ = 'sint'
declare *hol* target_rep function integerFromInt$_{64}$ = 'w2int'
declare *coq* target_rep function integerFromInt$_{64}$ = ''

assert *integer_from_int*$_{64}$_0 : integerFromInt$_{64}$ 0 = 0
assert *integer_from_int*$_{64}$_1 : integerFromInt$_{64}$ 1 = 1
assert *integer_from_int*$_{64}$_2 : integerFromInt$_{64}$ 123 = 123
assert *integer_from_int*$_{64}$_3 : integerFromInt$_{64}$ (−0) = −0
assert *integer_from_int*$_{64}$_4 : integerFromInt$_{64}$ (−1) = −1
assert *integer_from_int*$_{64}$_5 : integerFromInt$_{64}$ (−123) = −123

(* * * * * * * * * * * * * * * * * *)
(* naturalFrom... *)
(* * * * * * * * * * * * * * * * * *)

val *naturalFromNat* : NAT → $\MATHBB{N}$
declare *hol* target_rep function naturalFromNat *x* = (''*x* : $\MATHBB{N}$)
declare *ocaml* target_rep function naturalFromNat = 'Nat_big_num.of_int'
declare *isabelle* target_rep function naturalFromNat = ''
declare *coq* target_rep function naturalFromNat = ''

assert *natural_from_nat*$_0$: naturalFromNat 0 = 0
assert *natural_from_nat*$_1$: naturalFromNat 1 = 1
assert *natural_from_nat*$_2$: naturalFromNat 2 = 2

val *naturalFromInteger* : $\MATHBB{Z}$ → $\MATHBB{N}$
declare compile_message naturalFromInteger = *"naturalFromInteger is undefined for negative integers"*

declare *hol* target_rep function naturalFromInteger *i* = 'Num' ('ABS' *i*)
declare *ocaml* target_rep function naturalFromInteger = 'Nat_big_num.abs'
declare *coq* target_rep function naturalFromInteger = 'Z.abs_nat'
declare *isabelle* target_rep function naturalFromInteger *i* = 'nat' ('abs' *i*)

assert *natural_from_integer*$_0$: naturalFromInteger 0 = 0
assert *natural_from_integer*$_1$: naturalFromInteger 1 = 1
assert *natural_from_integer*$_2$: naturalFromInteger (− 2) = 2

(* * * * * * * * * * * * * * * * * *)

```
(* intFrom ...  *)
(* * * * * * * * * * * * * * * * * *)

val intFromInteger : $\MATHBB{Z}$ → INT
declare compile_message naturalFromInteger = "naturalFromInteger is undefined for negative integers and might fail for n

declare hol target_rep function intFromInteger = 'I' (* remove natFromNumeral, as it is the identify function *)
declare ocaml target_rep function intFromInteger = 'Nat_big_num.to_int'
declare isabelle target_rep function intFromInteger = ''
declare coq target_rep function intFromInteger = ''

assert int_from_integer_0 : intFromInteger 0 = 0
assert int_from_integer_1 : intFromInteger 1 = 1
assert int_from_integer_2 : intFromInteger (−2) = (−2)

val intFromNat : NAT → INT
declare hol target_rep function intFromNat = 'int_of_num'
declare ocaml target_rep function intFromNat n = ''n
declare isabelle target_rep function intFromNat = 'int'
declare coq target_rep function intFromNat n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n)))

assert int_from_nat_0 : intFromNat 0 = 0
assert int_from_nat_1 : intFromNat 1 = 1
assert int_from_nat_2 : intFromNat 2 = 2


(* * * * * * * * * * * * * * * * * *)
(* natFrom ...  *)
(* * * * * * * * * * * * * * * * * *)

val natFromNatural : $\MATHBB{N}$ → NAT
declare compile_message naturalFromInteger = "x natFromNatural might fail for too big values. The values allowed are sys

declare hol target_rep function natFromNatural x = (''x : NAT)
declare ocaml target_rep function natFromNatural = 'Nat_big_num.to_int'
declare isabelle target_rep function natFromNatural = ''
declare coq target_rep function natFromNatural = ''

assert nat_from_natural_0 : natFromNatural 0 = 0
assert nat_from_natural_1 : natFromNatural 1 = 1
assert nat_from_natural_2 : natFromNatural 2 = 2

val natFromInt : INT → NAT
declare hol target_rep function natFromInt i = 'Num' ('ABS' i)
declare ocaml target_rep function natFromInt = 'abs'
declare coq target_rep function natFromInt = 'Z.abs_nat'
declare isabelle target_rep function natFromInt i = 'nat' ('abs' i)

assert nat_from_int_0 : natFromInt 0 = 0
assert nat_from_int_1 : natFromInt 1 = 1
assert nat_from_int_2 : natFromInt (− 2) = 2


(* * * * * * * * * * * * * * * * * *)
(* int32From ...  *)
(* * * * * * * * * * * * * * * * * *)

val int32FromNat : NAT → INT32
```

```
declare hol target_rep function int32FromNat n = (('n2w' n) : INT32)
declare ocaml target_rep function int32FromNat = 'Int32.of_int'
declare coq target_rep function int32FromNat n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))) (* TODO check *)
declare isabelle target_rep function int32FromNat n = (('word_of_int' ('int' n)) : INT32)

assert int32_from_nat_0 : int32FromNat 0 = 0
assert int32_from_nat_1 : int32FromNat 1 = 1
assert int32_from_nat_2 : int32FromNat 123 = 123

val int32FromNatural : $\MATHBB{N}$ → INT32
declare hol target_rep function int32FromNatural n = (('n2w' n) : INT32)
declare ocaml target_rep function int32FromNatural = 'Nat_big_num.to_int'32
declare coq target_rep function int32FromNatural n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))) (* TODO check *)
declare isabelle target_rep function int32FromNatural n = (('word_of_int' ('int' n)) : INT32)

assert int32_from_natural_0 : int32FromNatural 0 = 0
assert int32_from_natural_1 : int32FromNatural 1 = 1
assert int32_from_natural_2 : int32FromNatural 123 = 123

val int32FromInteger : $\MATHBB{Z}$ → INT32
let int32FromInteger i = (
  let abs_int32 = int32FromNatural (naturalFromInteger i) in
  if (i < 0) then (- abs_int32) else abs_int32
)

declare ocaml target_rep function int32FromInteger = 'Nat_big_num.to_int'32
declare isabelle target_rep function int32FromInteger i = (('word_of_int' i) : INT32)

assert int32_from_integer_0 : int32FromInteger 0 = 0
assert int32_from_integer_1 : int32FromInteger 1 = 1
assert int32_from_integer_2 : int32FromInteger 123 = 123
assert int32_from_integer_3 : int32FromInteger (-0) = -0
assert int32_from_integer_4 : int32FromInteger (-1) = -1
assert int32_from_integer_5 : int32FromInteger (-123) = -123

val int32FromInt : INT → INT32
let int32FromInt i = int32FromInteger (integerFromInt i)
declare ocaml target_rep function int32FromInt = 'Int32.of_int'
declare isabelle target_rep function int32FromInt i = (('word_of_int' i) : INT32)

assert int32_from_int_0 : int32FromInt 0 = 0
assert int32_from_int_1 : int32FromInt 1 = 1
assert int32_from_int_2 : int32FromInt 123 = 123
assert int32_from_int_3 : int32FromInt (-0) = -0
assert int32_from_int_4 : int32FromInt (-1) = -1
assert int32_from_int_5 : int32FromInt (-123) = -123


val int32FromInt64 : INT64 → INT32
let int32FromInt64 i = int32FromInteger (integerFromInt64 i)
declare ocaml target_rep function int32FromInt64 = 'Int64.to_int'32
declare hol target_rep function int32FromInt64 i = (('sw2sw' i) : INT32)
declare isabelle target_rep function int32FromInt64 i = (('scast' i) : INT32)

assert int32_from_int64_0 : int32FromInt64 0 = 0
assert int32_from_int64_1 : int32FromInt64 1 = 1
assert int32_from_int64_2 : int32FromInt64 123 = 123
assert int32_from_int64_3 : int32FromInt64 (-0) = -0
```

assert $int32\_from\_int_{64}\_4$: int32FromInt$_{64}$ (-1) = -1
assert $int32\_from\_int_{64}\_5$: int32FromInt$_{64}$ (-123) = -123

(* * * * * * * * * * * * * * * * * *)
(* int64From ... *)
(* * * * * * * * * * * * * * * * * *)

val *int64FromNat* : NAT → INT$_{64}$
declare *hol* target_rep function int64FromNat n = (('n2w' n) : INT$_{64}$)
declare *ocaml* target_rep function int64FromNat = 'Int64.of_int'
declare *coq* target_rep function int64FromNat n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))) (* TODO check *)
declare *isabelle* target_rep function int64FromNat n = (('word_of_int' ('int' n)) : INT$_{64}$)

assert $int64\_from\_nat_0$: int64FromNat 0 = 0
assert $int64\_from\_nat_1$: int64FromNat 1 = 1
assert $int64\_from\_nat_2$: int64FromNat 123 = 123

val *int64FromNatural* : $\MATHBB{N}$ → INT$_{64}$
declare *hol* target_rep function int64FromNatural n = (('n2w' n) : INT$_{64}$)
declare *ocaml* target_rep function int64FromNatural = 'Nat_big_num.to_int'$_{64}$
declare *coq* target_rep function int64FromNatural n = ('Z.pred' ('Z.pos' ('P_of_succ_nat' n))) (* TODO check *)
declare *isabelle* target_rep function int64FromNatural n = (('word_of_int' ('int' n)) : INT$_{64}$)

assert $int64\_from\_natural_0$: int64FromNatural 0 = 0
assert $int64\_from\_natural_1$: int64FromNatural 1 = 1
assert $int64\_from\_natural_2$: int64FromNatural 123 = 123

val *int64FromInteger* : $\MATHBB{Z}$ → INT$_{64}$
let *int64FromInteger* i = (
  let abs_int_{64} = int64FromNatural (naturalFromInteger i) in
  if $(i < 0)$ then $(-\ abs\_int_{64})$ else abs_int_{64}
)

declare *ocaml* target_rep function int64FromInteger = 'Nat_big_num.to_int'$_{64}$
declare *isabelle* target_rep function int64FromInteger i = (('word_of_int' i) : INT$_{64}$)

assert $int64\_from\_integer_0$: int64FromInteger 0 = 0
assert $int64\_from\_integer_1$: int64FromInteger 1 = 1
assert $int64\_from\_integer_2$: int64FromInteger 123 = 123
assert $int64\_from\_integer_3$: int64FromInteger (-0) = -0
assert $int64\_from\_integer_4$: int64FromInteger (-1) = -1
assert $int64\_from\_integer_5$: int64FromInteger (-123) = -123

val *int64FromInt* : INT → INT$_{64}$
let *int64FromInt* i = int64FromInteger (integerFromInt i)
declare *ocaml* target_rep function int64FromInt = 'Int64.of_int'
declare *isabelle* target_rep function int64FromInt i = (('word_of_int' i) : INT$_{64}$)

assert $int64\_from\_int_0$: int64FromInt 0 = 0
assert $int64\_from\_int_1$: int64FromInt 1 = 1
assert $int64\_from\_int_2$: int64FromInt 123 = 123
assert $int64\_from\_int_3$: int64FromInt (-0) = -0
assert $int64\_from\_int_4$: int64FromInt (-1) = -1
assert $int64\_from\_int_5$: int64FromInt (-123) = -123

val $int64FromInt_{32}$: INT_{32} → INT_{64}
let $int64FromInt_{32}$ i = int64FromInteger (integerFromInt$_{32}$ i)
declare *ocaml* target_rep function int64FromInt$_{32}$ = `'Int64.of_int'`$_{32}$
declare *hol* target_rep function int64FromInt$_{32}$ i = ((`'sw2sw'` i) : INT_{64})
declare *isabelle* target_rep function int64FromInt$_{32}$ i = ((`'scast'` i) : INT_{64})

assert $int64\_from\_int_{33}\_0$: int64FromInt$_{32}$ 0 = 0
assert $int64\_from\_int_{32}\_1$: int64FromInt$_{32}$ 1 = 1
assert $int64\_from\_int_{32}\_2$: int64FromInt$_{32}$ 123 = 123
assert $int64\_from\_int_{32}\_3$: int64FromInt$_{32}$ (−0) = −0
assert $int64\_from\_int_{32}\_4$: int64FromInt$_{32}$ (−1) = −1
assert $int64\_from\_int_{32}\_5$: int64FromInt$_{32}$ (−123) = −123

(* * * * * * * * * * * * * * * * * *)
(* `what's missing` *)
(* * * * * * * * * * * * * * * * * *)

val *naturalFromInt* : INT → $\mathbb{N}$
val $naturalFromInt_{32}$: INT_{32} → $\mathbb{N}$
val $naturalFromInt_{64}$: INT_{64} → $\mathbb{N}$

let inline *naturalFromInt* i = naturalFromNat (natFromInt i)
let inline $naturalFromInt_{32}$ i = naturalFromInteger (integerFromInt$_{32}$ i)
let inline $naturalFromInt_{64}$ i = naturalFromInteger (integerFromInt$_{64}$ i)

assert $natural\_from\_int_{0}$: naturalFromInt 0 = 0
assert $natural\_from\_int_{1}$: naturalFromInt 1 = 1
assert $natural\_from\_int_{2}$: naturalFromInt (− 2) = 2
assert $natural\_from\_int_{32}\_0$: naturalFromInt$_{32}$ 0 = 0
assert $natural\_from\_int_{32}\_1$: naturalFromInt$_{32}$ 1 = 1
assert $natural\_from\_int_{32}\_2$: naturalFromInt$_{32}$ (− 2) = 2
assert $natural\_from\_int_{64}\_0$: naturalFromInt$_{64}$ 0 = 0
assert $natural\_from\_int_{64}\_1$: naturalFromInt$_{64}$ 1 = 1
assert $natural\_from\_int_{64}\_2$: naturalFromInt$_{64}$ (− 2) = 2

val *intFromNatural* : $\mathbb{N}$ → INT
val $intFromInt_{32}$: INT_{32} → INT
val $intFromInt_{64}$: INT_{64} → INT

let inline *intFromNatural* n = intFromNat (natFromNatural n)
let inline $intFromInt_{32}$ i = intFromInteger (integerFromInt$_{32}$ i)
let inline $intFromInt_{64}$ i = intFromInteger (integerFromInt$_{64}$ i)

assert $int\_from\_natural_{0}$: intFromNatural 0 = 0
assert $int\_from\_natural_{1}$: intFromNatural 1 = 1
assert $int\_from\_natural_{2}$: intFromNatural 122 = 122
assert $int\_from\_int_{32}\_0$: intFromInt$_{32}$ 0 = 0
assert $int\_from\_int_{32}\_1$: intFromInt$_{32}$ 1 = 1
assert $int\_from\_int_{32}\_2$: intFromInt$_{32}$ (− 2) = (−2)
assert $int\_from\_int_{64}\_0$: intFromInt$_{64}$ 0 = 0
assert $int\_from\_int_{64}\_1$: intFromInt$_{64}$ 1 = 1
assert $int\_from\_int_{64}\_2$: intFromInt$_{64}$ (− 2) = (−2)

val *natFromInteger* : $\mathbb{Z}$ → NAT
val $natFromInt_{32}$: INT_{32} → NAT

val $natFromInt_{64}$: $\text{INT}_{64} \rightarrow \text{NAT}$

let inline $natFromInteger\ n = \text{natFromInt}\ (\text{intFromInteger}\ n)$
let inline $natFromInt_{32}\ i = \text{natFromInteger}\ (\text{integerFromInt}_{32}\ i)$
let inline $natFromInt_{64}\ i = \text{natFromInteger}\ (\text{integerFromInt}_{64}\ i)$

assert $nat\_from\_integer_0$: $\text{natFromInteger}\ 0 = 0$
assert $nat\_from\_integer_1$: $\text{natFromInteger}\ 1 = 1$
assert $nat\_from\_integer_2$: $\text{natFromInteger}\ 122 = 122$
assert $nat\_from\_int_{32}\_0$: $\text{natFromInt}_{32}\ 0 = 0$
assert $nat\_from\_int_{32}\_1$: $\text{natFromInt}_{32}\ 1 = 1$
assert $nat\_from\_int_{32}\_2$: $\text{natFromInt}_{32}\ (-\ 2) = 2$
assert $nat\_from\_int_{64}\_0$: $\text{natFromInt}_{64}\ 0 = 0$
assert $nat\_from\_int_{64}\_1$: $\text{natFromInt}_{64}\ 1 = 1$
assert $nat\_from\_int_{64}\_2$: $\text{natFromInt}_{64}\ (-\ 2) = 2$

6 Tuple

```
(* ********************************************************************************
(* Tuples                                                                         *)
(* ********************************************************************************

(* The type for tuples (pairs) is hard-coded, so here only a few functions are used *)

declare {isabelle, hol, ocaml, coq} rename module = lem_tuple

open import Bool Basic_classes

(* ------------------------- *)
(* fst                       *)
(* ------------------------- *)

val fst : ∀ α β. α * β → α
let fst (v1, v2) = v1

declare hol target_rep function fst = 'FST'
declare ocaml target_rep function fst = 'fst'
declare isabelle target_rep function fst = 'fst'
declare coq target_rep function fst = ('@' 'fst' '_' '_')

assert fst1 : (fst (true, false) = true)
assert fst2 : (fst (false, true) = false)

(* ------------------------- *)
(* snd                       *)
(* ------------------------- *)

val snd : ∀ α β. α * β → β
let snd (v1, v2) = v2

declare hol target_rep function snd = 'SND'
declare ocaml target_rep function snd = 'snd'
declare isabelle target_rep function snd = 'snd'
declare coq target_rep function snd = ('@' 'snd' '_' '_')

lemma fst_snd : (∀ v. v = (fst v, snd v))

assert snd1 : (snd (true, false) = false)
assert snd2 : (snd (false, true) = true)


(* ------------------------- *)
(* curry                     *)
(* ------------------------- *)

val curry : ∀ α β γ. (α * β → γ) → (α → β → γ)
let inline curry f v1 v2 = f (v1, v2)

declare hol target_rep function curry = 'CURRY'
declare isabelle target_rep function curry = 'curry'
declare ocaml target_rep function curry = 'Lem.curry'
declare coq target_rep function curry = 'prod_curry'

assert curry1 : (curry (fun (x, y) → x ∧ y) true false = false)
```

```
(* ------------------------------------------------- *)
(* uncurry                                           *)
(* ------------------------------------------------- *)

val uncurry : ∀ α β γ. (α → β → γ) → (α * β → γ)
let inline uncurry f = (fun (v1, v2) → f v1 v2)

declare hol target_rep function uncurry = 'UNCURRY'
declare isabelle target_rep function uncurry = 'case_prod'
declare ocaml target_rep function uncurry = 'Lem.uncurry'
declare coq target_rep function uncurry = 'prod_uncurry'

lemma curry_uncurry : (∀ f xy. uncurry (curry f) xy = f xy)
lemma uncurry_curry : (∀ f x y. curry (uncurry f) x y = f x y)

assert uncurry_1 : (uncurry (fun x y → x ∧ y) (true, false) = false)


(* ------------------------------------------------- *)
(* swap                                              *)
(* ------------------------------------------------- *)

val swap : ∀ α β. (α * β) → (β * α)
let swap (v1, v2) = (v2, v1)

let inline {isabelle, coq} swap = (fun (v1, v2) → (v2, v1))
declare hol target_rep function swap = 'SWAP'
declare ocaml target_rep function swap = 'Lem.pair_swap'

assert swap_1 : (swap (false, true) = (true, false))
```

7 List

```
(*****************************************************************************
(* A library for lists                                                       *)
(*                                                                           *)
(* It mainly follows the Haskell List - library                              *)
(*****************************************************************************



(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)


declare {isabelle, ocaml, hol, coq} rename module = lem_list

open import Bool Maybe Basic_classes Function Tuple Num

open import {coq} Coq.Lists.List
open import {isabelle} $LIB_DIR/Lem
open import {hol} lemTheory listTheory rich_listTheory sortingTheory

(* ========================================================================== *)
(* Basic list functions                                                       *)
(* ========================================================================== *)

(* The type of lists as well as list literals like [], [1;2], ... are hardcoded.  Thus, we can directly dive into


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* cons                                                                      *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val :: : ∀ α. α → LIST α → LIST α

declare ascii_rep function :: = cons
declare hol target_rep function cons = infix '::'
declare ocaml target_rep function cons = infix '::'
declare isabelle target_rep function cons = infix '#'
declare coq target_rep function cons = infix '::'


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* Emptyness check                                                           *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val null : ∀ α. LIST α → BOOL
let null l = match l with [] → true | _ → false end

declare hol target_rep function null = 'NULL'
declare {ocaml} rename function null = list_null
let inline {isabelle} null l = (l = [])

assert null_simple1 : (null ([] : LIST NAT))
assert null_simple2 : (¬ (null [(2 : NAT); 3; 4]))
assert null_simple3 : (¬ (null [(2 : NAT)]))
```

```
(* ---------------------------------- *)
(* Length                             *)
(* ---------------------------------- *)

val length : ∀ α. LIST α → NAT
let rec length l =
  match l with
  | [] → 0
  | x :: xs → length xs + 1
 end

declare termination_argument length = automatic

declare hol target_rep function length = 'LENGTH'
declare ocaml target_rep function length = 'List.length'
declare isabelle target_rep function length = 'List.length'
declare coq target_rep function length = 'List.length'

assert length0 : (length ([] : LIST NAT) = 0)
assert length1 : (length ([2] : LIST NAT) = 1)
assert length2 : (length ([2; 3] : LIST NAT) = 2)

lemma length_spec : ((length [] = 0) ∧ (∀ x xs. length (x :: xs) = length xs + 1))

(* ---------------------------------- *)
(* Equality                           *)
(* ---------------------------------- *)

val listEqual : ∀ α. Eq α ⇒ LIST α → LIST α → BOOL
val listEqualBy : ∀ α. (α → α → BOOL) → LIST α → LIST α → BOOL

let rec listEqualBy eq l1 l2 = match (l1, l2) with
  | ([], []) → true
  | ([], (_ :: _)) → false
  | ((_ :: _), []) → false
  | (x :: xs, y :: ys) → (eq x y ∧ listEqualBy eq xs ys)
end
declare termination_argument listEqualBy = automatic

let inline listEqual = listEqualBy (=)
declare hol target_rep function listEqual = infix '='
declare isabelle target_rep function listEqual = infix '='
declare coq target_rep function listEqualBy = 'list_equal_by'

instance ∀ α. Eq α ⇒ (Eq (LIST α))
  let = = listEqual
  let <> l1 l2 = ¬ (listEqual l1 l2)
end


(* ---------------------------------- *)
(* compare                            *)
(* ---------------------------------- *)

val lexicographicCompare : ∀ α. Ord α ⇒ LIST α → LIST α → ORDERING
val lexicographicCompareBy : ∀ α. (α → α → ORDERING) → LIST α → LIST α → ORDERING

let rec lexicographicCompareBy cmp l1 l2 = match (l1, l2) with
```

```
  | ([], []) → EQ
  | ([], _ :: _) → LT
  | (_ :: _, []) → GT
  | (x :: xs, y :: ys) → begin
      match cmp x y with
        | LT → LT
        | GT → GT
        | EQ → lexicographicCompareBy cmp xs ys
      end
    end
end
declare termination_argument lexicographicCompareBy = automatic

let inline lexicographicCompare = lexicographicCompareBy compare
declare {ocaml, hol} rename function lexicographicCompareBy = lexicographic_compare

val lexicographicLess : ∀ α. Ord α ⇒ LIST α → LIST α → BOOL
val lexicographicLessBy : ∀ α. (α → α → BOOL) → (α → α → BOOL) → LIST α → LIST α → BOOL

let rec lexicographicLessBy less less_eq l1 l2 = match (l1, l2) with
  | ([], []) → false
  | ([], _ :: _) → true
  | (_ :: _, []) → false
  | (x :: xs, y :: ys) → ((less x y) ∨ ((less_eq x y) ∧ (lexicographicLessBy less less_eq xs ys)))
end
declare termination_argument lexicographicLessBy = automatic

let inline lexicographicLess = lexicographicLessBy (<) (≤)
declare {ocaml, hol} rename function lexicographicLessBy = lexicographic_less

val lexicographicLessEq : ∀ α. Ord α ⇒ LIST α → LIST α → BOOL
val lexicographicLessEqBy : ∀ α. (α → α → BOOL) → (α → α → BOOL) → LIST α → LIST α →
BOOL
let rec lexicographicLessEqBy less less_eq l1 l2 = match (l1, l2) with
  | ([], []) → true
  | ([], _ :: _) → true
  | (_ :: _, []) → false
  | (x :: xs, y :: ys) → (less x y ∨ (less_eq x y ∧ lexicographicLessEqBy less less_eq xs ys))
end
declare termination_argument lexicographicLessEqBy = automatic

let inline lexicographicLessEq = lexicographicLessEqBy (<) (≤)
declare {ocaml, hol} rename function lexicographicLessEqBy = lexicographic_less_eq


instance ∀ α. Ord α ⇒ (Ord (LIST α))
  let compare = lexicographicCompare
  let < = lexicographicLess
  let <= = lexicographicLessEq
  let > x y = lexicographicLess y x
  let >= x y = lexicographicLessEq y x
end


assert list_ord1 : ([] < [(2 : NAT)])
assert list_ord2 : ([] ≤ [(2 : NAT)])
assert list_ord3 : ([1] ≤ [(2 : NAT)])
assert list_ord4 : ([2] ≤ [(2 : NAT)])
```

```
assert list_ord5 : ([2;3] > [(2 : NAT)])
assert list_ord6 : ([2;3;4;5] > [(2 : NAT)])
assert list_ord7 : ([2;3;4] > [(2 : NAT);1;5;67])
assert list_ord8 : ([4] > [(3 : NAT);56])
assert list_ord9 : ([5] ≥ [(5 : NAT)])


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* Append                                                         *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val ++ : ∀ α. LIST α → LIST α → LIST α (* originally append *)
let rec ++ xs ys = match xs with
                   | [] → ys
                   | x :: xs' → x :: (append xs' ys)
                 end

declare ascii_rep function ++ = append
declare termination_argument append = automatic

declare hol target_rep function append = infix '++'
declare ocaml target_rep function append l1 l2 = 'List.rev_append' ('List.rev' l1) l2
declare isabelle target_rep function append = infix '@'
declare tex target_rep function append = infix '$+\!+$'
declare coq target_rep function append = ('@' 'List.app' '_')

assert append1 : ([0;1;2;3] ++ [4;5] = [(0 : NAT);1;2;3;4;5])
lemma append_nil1 : (∀ l. l ++ [] = l)
lemma append_nil2 : (∀ l. [] ++ l = l)

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* snoc                                                           *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val snoc : ∀ α. α → LIST α → LIST α
let snoc e l = l ++ [e]

declare hol target_rep function snoc = 'SNOC'
let inline {isabelle, coq} snoc e l = l ++ [e]

assert snoc1 : snoc (2 : NAT) [] = [2]
assert snoc2 : snoc (2 : NAT) [3;4] = [3;4;2]
assert snoc3 : snoc (2 : NAT) [1] = [1;2]
lemma snoc_length : ∀ e l. length (snoc e l) = succ (length l)
lemma snoc_append : ∀ e l1 l2. (snoc e (l1 ++ l2) = l1 ++ (snoc e l2))


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* Reverse                                                        *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* First lets define the function [reverse_append], which is  closely related to reverse. [reverse_append l1 l

val reverseAppend : ∀ α. LIST α → LIST α → LIST α (* originally named rev_append *)
let rec reverseAppend l1 l2 = match l1 with
                              | [] → l2
                              | x :: xs → reverseAppend xs (x :: l2)
                              end
```

```
declare termination_argument reverseAppend = automatic

declare hol target_rep function reverseAppend = 'REV'
declare ocaml target_rep function reverseAppend = 'List.rev_append'

assert reverseAppend_1 : (reverseAppend [(0 : NAT); 1; 2; 3] [4; 5] = [3; 2; 1; 0; 4; 5])

(* Reversing a list *)
val reverse : ∀ α. LIST α → LIST α (* originally named rev *)
let reverse l = reverseAppend l []

declare hol target_rep function reverse = 'REVERSE'
declare ocaml target_rep function reverse = 'List.rev'
declare isabelle target_rep function reverse = 'List.rev'
declare coq target_rep function reverse = 'List.rev'

assert reverse_nil : (reverse ([] : LIST NAT) = [])
assert reverse_1 : (reverse [(1 : NAT)] = [1])
assert reverse_2 : (reverse [(1 : NAT); 2] = [2; 1])
assert reverse_5 : (reverse [(1 : NAT); 2; 3; 4; 5] = [5; 4; 3; 2; 1])

lemma reverseAppend : (∀ l_1 l_2. reverseAppend l_1 l_2 = (++) (reverse l_1) l_2)
let inline {isabelle} reverseAppend l_1 l_2 = ((reverse l_1) ++ l_2)

(* ------------------------- *)
(* Map                       *)
(* ------------------------- *)

val map_tr : ∀ α β. LIST β → (α → β) → LIST α → LIST β
let rec map_tr rev_acc f l = match l with
  | [] → reverse rev_acc
  | x :: xs → map_tr ((f x) :: rev_acc) f xs
end

(* taken from : https://blogs.janestreet.com/optimizing-list-map/ *)
val count_map : ∀ α β. (α → β) → LIST α → NAT → LIST β
let rec count_map f l ctr =
   match l with
  | [] → []
  | hd :: tl → f hd ::
    (if ctr < 5000 then count_map f tl (ctr + 1)
    else map_tr [] f tl)
  end

val map : ∀ α β. (α → β) → LIST α → LIST β
let map f l = count_map f l 0

declare termination_argument map = automatic

declare hol target_rep function map = 'MAP'
(** DPM: for standard List.map replace line below, otherwise uses imperative * version supplied with Lem t
(*declare ocaml   target_rep function map = `List.map`*)
declare isabelle target_rep function map = 'List.map'
declare coq target_rep function map = 'List.map'

assert map_nil : (map (fun x → x + (1 : NAT)) [] = [])
assert map_1 : (map (fun x → x + (1 : NAT)) [0] = [1])
assert map_2 : (map (fun x → x + (1 : NAT)) [0; 1] = [1; 2])
```

```
assert map_3 : (map (fun x → x + (1 : NAT)) [0; 1; 2] = [1; 2; 3])
assert map_4 : (map (fun x → x + (1 : NAT)) [0; 1; 2; 3] = [1; 2; 3; 4])
assert map_5 : (map (fun x → x + (1 : NAT)) [0; 1; 2; 3; 4] = [1; 2; 3; 4; 5])
assert map_6 : (map (fun x → x + (1 : NAT)) [0; 1; 2; 3; 4; 5] = [1; 2; 3; 4; 5; 6])

(* ------------------------- *)
(* Reverse Map               *)
(* ------------------------- *)

val reverseMap : ∀ α β. (α → β) → LIST α → LIST β
let inline reverseMap f l = reverse (map f l)

declare ocaml target_rep function reverseMap = 'List.rev_map'


(* ========================================================================== *)
(* Folding                                                                    *)
(* ========================================================================== *)

(* ------------------------- *)
(* fold left                 *)
(* ------------------------- *)

val foldl : ∀ α β. (α → β → α) → α → LIST β → α  (* originally foldl *)

let rec foldl f b l =  match l with
  | [] → b
  | x :: xs → foldl f (f b x) xs
end
declare termination_argument foldl = automatic

declare hol target_rep function foldl = 'FOLDL'
declare ocaml target_rep function foldl = 'List.fold_left'
declare isabelle target_rep function foldl = 'List.foldl'
declare coq target_rep function foldl f e l = 'List.fold_left' f l e

assert foldl_0 : (foldl (+) (0 : NAT) [] = 0)
assert foldl_1 : (foldl (+) (0 : NAT) [4] = 4)
assert foldl_4 : (foldl (fun l e → e::l) [] [(1 : NAT); 2; 3; 4] = [4; 3; 2; 1])


(* ------------------------- *)
(* fold right                *)
(* ------------------------- *)

val foldr : ∀ α β. (α → β → β) → β → LIST α → β (* originally foldr with different argument order *)
let rec foldr f b l =  match l with
  | [] → b
  | x :: xs → f x (foldr f b xs)
end
declare termination_argument foldr = automatic

declare hol target_rep function foldr = 'FOLDR'
declare ocaml target_rep function foldr f b l = 'List.fold_right' f l b
declare isabelle target_rep function foldr f b l = 'List.foldr' f l b
declare coq target_rep function foldr = 'List.fold_right'

assert foldr_0 : (foldr (+) (0 : NAT) [] = 0)
```

```
assert foldr_1 : (foldr (+) 1 [(4 : NAT)] = 5)
assert foldr_4 : (foldr (fun e l → e::l) [] [(1 : NAT); 2; 3; 4] = [1; 2; 3; 4])


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* concatenating lists    *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val concat : ∀ α. LIST (LIST α) → LIST α  (* before also called "flatten" *)
let concat =  foldr (++) []

declare hol target_rep function concat = 'FLAT'
declare ocaml target_rep function concat = 'List.concat'
declare isabelle target_rep function concat = 'List.concat'

assert concat_nil : (concat ([] : LIST (LIST NAT)) = [])
assert concat_1 : (concat [[(1 : NAT)]] = [1])
assert concat_2 : (concat [[(1 : NAT)]; [2]] = [1; 2])
assert concat_3 : (concat [[(1 : NAT)]; []; [2]] = [1; 2])

lemma concat_emp_thm : (concat [] = [])
lemma concat_cons_thm : (∀ l ll. (concat (l::ll) = (++) l (concat ll)))


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* concatenating with mapping *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val concatMap : ∀ α β. (α → LIST β) → LIST α → LIST β
let inline concatMap f l = concat (map f l)

assert concatMap_nil : (concatMap (fun (x : NAT) → [x; x]) [] = [])
assert concatMap_1 : (concatMap (fun x → [x; x]) [(1 : NAT)] = [1; 1])
assert concatMap_2 : (concatMap (fun x → [x; x]) [(1 : NAT); 2] = [1; 1; 2; 2])
assert concatMap_3 : (concatMap (fun x → [x; x]) [(1 : NAT); 2; 3] = [1; 1; 2; 2; 3; 3])
lemma concatMap_concat : (∀ ll. concat ll = concatMap (fun l → l) ll)
lemma concatMap_alt_def : (∀ f l. concatMap f l = foldr (fun l ll → f l ++ ll) [] l)


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* universal qualification  *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val all : ∀ α. (α → BOOL) → LIST α → BOOL  (* originally for_all *)
let all P l =  foldl (fun r e → P e ∧ r) true l

declare hol target_rep function all = 'EVERY'
declare ocaml target_rep function all = 'List.for_all'
declare isabelle target_rep function all P l = (∀ x ∈ ('set' l). P x)
declare coq target_rep function all = 'List.forallb'

assert all_0 : (all (fun x → x > (2 : NAT)) [])
assert all_4 : (all (fun x → x > (2 : NAT)) [4; 5; 6; 7])
assert all_4_neg : (¬ (all (fun x → x > (2 : NAT)) [4; 5; 2; 7]))

lemma all_nil_thm : (∀ P. all P [])
lemma all_cons_thm : (∀ P e l. all P (e::l) = (P e ∧ all P l))
```

```
(* ------------------------------------------ *)
(* existential qualification *)
(* ------------------------------------------ *)

val any : ∀ α. (α → BOOL) → LIST α → BOOL (* originally exist *)
let any P l = foldl (fun r e → P e ∨ r) false l

declare hol target_rep function any = 'EXISTS'
declare ocaml target_rep function any = 'List.exists'
declare isabelle target_rep function any P l = (∃ x ∈ ('set' l). P x)
declare coq target_rep function any = 'List.existsb'

assert any_0 : (¬ (any (fun x → (x < (3 : NAT))) []))
assert any_4 : (¬ (any (fun x → (x < (3 : NAT))) [4;5;6;7]))
assert any_4_neg : (any (fun x → (x < (3 : NAT))) [4;5;2;7])

lemma any_nil_thm : (∀ P. ¬ (any P []))
lemma any_cons_thm : (∀ P e l. any P (e::l) = (P e ∨ any P l))


(* ------------------------------------------ *)
(* dest_init                                  *)
(* ------------------------------------------ *)

(* get the initial part and the last element of the list in a safe way *)

val dest_init : ∀ α. LIST α → MAYBE (LIST α * α)

let rec dest_init_aux rev_init last_elem_seen to_process =
   match to_process with
    | [] → (reverse rev_init, last_elem_seen)
    | x :: xs → dest_init_aux (last_elem_seen::rev_init) x xs
  end
declare termination_argument dest_init_aux = automatic

let dest_init l = match l with
  | [] → Nothing
  | x :: xs → Just (dest_init_aux [] x xs)
end

assert dest_init_0 : (dest_init ([] : LIST NAT) = Nothing)
assert dest_init_1 : (dest_init [(1 : NAT)] = Just ([], 1))
assert dest_init_2 : (dest_init [(1 : NAT);2;3;4;5] = Just ([1;2;3;4], 5))

lemma dest_init_nil : (dest_init [] = Nothing)
lemma dest_init_snoc : (∀ x xs. dest_init (xs ++ [x]) = Just (xs, x))


(* ========================================================================== *)
(* Indexing lists                                                             *)
(* ========================================================================== *)

(* ------------------------------------------ *)
(* index / nth with maybe                     *)
(* ------------------------------------------ *)
```

```
val index : ∀ α. LIST α → NAT → MAYBE α

let rec index l n = match l with
  | [] → Nothing
  | x :: xs → if n = 0 then Just x else index xs (n−1)
end

declare termination_argument index = automatic

declare isabelle target_rep function index = 'index'
declare {ocaml, hol} rename function index = list_index

assert index_0 : (index [(0 : NAT); 1; 2; 3; 4; 5] 0 = Just 0)
assert index_1 : (index [(0 : NAT); 1; 2; 3; 4; 5] 1 = Just 1)
assert index_2 : (index [(0 : NAT); 1; 2; 3; 4; 5] 2 = Just 2)
assert index_3 : (index [(0 : NAT); 1; 2; 3; 4; 5] 3 = Just 3)
assert index_4 : (index [(0 : NAT); 1; 2; 3; 4; 5] 4 = Just 4)
assert index_5 : (index [(0 : NAT); 1; 2; 3; 4; 5] 5 = Just 5)
assert index_6 : (index [(0 : NAT); 1; 2; 3; 4; 5] 6 = Nothing)

lemma index_is_none : (∀ l n. (index l n = Nothing) ⟷ (n ≥ length l))
lemma index_list_eq : (∀ l1 l2. ((∀ n. index l1 n = index l2 n) ⟷ (l1 = l2)))

(* −−−−−−−−−−−−−−−−−−−−−−−−−−−−− *)
(* findIndices *)
(* −−−−−−−−−−−−−−−−−−−−−−−−−−−−− *)

(* [findIndices P l] returns the indices of all elements of list [l] that satisfy predicate [P].  Counting start
val findIndices : ∀ α. (α → BOOL) → LIST α → LIST NAT

let rec findIndices_aux (i : NAT) P l =
   match l with
    | [] → []
    | x :: xs → if P x then i :: findIndices_aux (i + 1) P xs else findIndices_aux (i + 1) P xs
 end
let findIndices P l = findIndices_aux 0 P l
declare termination_argument findIndices_aux = automatic

declare isabelle target_rep function findIndices = 'find_indices'
declare {ocaml, hol} rename function findIndices = find_indices
declare {ocaml, hol} rename function findIndices_aux = find_indices_aux

assert findIndices_1 : (findIndices (fun (n : NAT) → n > 3) [] = [])
assert findIndices_2 : (findIndices (fun (n : NAT) → n > 3) [4] = [0])
assert findIndices_3 : (findIndices (fun (n : NAT) → n > 3) [1; 5; 3; 1; 2; 6] = [1; 5])



(* −−−−−−−−−−−−−−−−−−−−−−−−−−−−− *)
(* findIndex *)
(* −−−−−−−−−−−−−−−−−−−−−−−−−−−−− *)

(* findIndex returns the first index of a list that satisfies a given predicate. *)
val findIndex : ∀ α. (α → BOOL) → LIST α → MAYBE NAT
let findIndex P l = match findIndices P l with
  | [] → Nothing
  | x :: _ → Just x
end
```

```
declare isabelle target_rep function findIndex = 'find_index'
declare {ocaml, hol} rename function findIndex = find_index

assert find_index_0 : (findIndex (fun (n : NAT) → n > 3) [1; 2] = Nothing)
assert find_index_1 : (findIndex (fun (n : NAT) → n > 3) [1; 2; 4] = Just 2)
assert find_index_2 : (findIndex (fun (n : NAT) → n > 3) [1; 2; 4; 5; 67; 1] = Just 2)

(* ------------------------------------------------------------------------ *)
(* elemIndices                                                              *)
(* ------------------------------------------------------------------------ *)

val elemIndices : ∀ α. Eq α ⇒ α → LIST α → LIST NAT
let inline elemIndices e l = findIndices ((=) e) l

assert elemIndices_0 : (elemIndices (2 : NAT) [] = [])
assert elemIndices_1 : (elemIndices (2 : NAT) [2] = [0])
assert elemIndices_2 : (elemIndices (2 : NAT) [2; 3; 4; 2; 4; 2] = [0; 3; 5])

(* ------------------------------------------------------------------------ *)
(* elemIndex                                                                *)
(* ------------------------------------------------------------------------ *)

val elemIndex : ∀ α. Eq α ⇒ α → LIST α → MAYBE NAT
let inline elemIndex e l = findIndex ((=) e) l

assert elemIndex_0 : (elemIndex (2 : NAT) [] = Nothing)
assert elemIndex_1 : (elemIndex (2 : NAT) [2] = Just 0)
assert elemIndex_2 : (elemIndex (2 : NAT) [3; 4; 2; 4; 2] = Just 2)


(* ========================================================================== *)
(* Creating lists                                                           *)
(* ========================================================================== *)

(* ------------------------------------------------------------------------ *)
(* genlist                                                                  *)
(* ------------------------------------------------------------------------ *)

(* [genlist f n] generates the list [f 1; ... (f (n - 1))] *)
val genlist : ∀ α. (NAT → α) → NAT → LIST α


let rec genlist f (n : NAT) =
  match (n : NAT) with
    | (0 : NAT) → []
    | n' + 1 → snoc (f n') (genlist f n')
  end
declare termination_argument genlist = automatic

assert genlist_0 : (genlist (fun n → n) 0 = [])
assert genlist_1 : (genlist (fun n → n) 1 = [0])
assert genlist_2 : (genlist (fun n → n) 2 = [0; 1])
assert genlist_3 : (genlist (fun n → n) 3 = [0; 1; 2])
lemma genlist_length : (∀ f n. (length (genlist f n) = n))
lemma genlist_index : (∀ f n i. i < n ⟶ index (genlist f n) i = Just (f i))
```

```
declare hol target_rep function genlist = 'GENLIST'
declare isabelle target_rep function genlist = 'genlist'


(* ------------------------------ *)
(* replicate                      *)
(* ------------------------------ *)

val replicate : ∀ α. NAT → α → LIST α
let rec replicate n x =
  match n with
   | 0 → []
   | n' + 1 → x :: replicate n' x
 end
declare termination_argument replicate = automatic

declare isabelle target_rep function replicate = 'List.replicate'
declare hol target_rep function replicate = 'REPLICATE'

assert replicate_0 : (replicate 0 (2 : NAT) = [])
assert replicate_1 : (replicate 1 (2 : NAT) = [2])
assert replicate_2 : (replicate 2 (2 : NAT) = [2; 2])
assert replicate_3 : (replicate 3 (2 : NAT) = [2; 2; 2])
lemma replicate_length : (∀ n x. (length (replicate n x) = n))
lemma replicate_index : (∀ n x i. i < n ⟶ index (replicate n x) i = Just x)


(* ============================================================================ *)
(* Sublists                                                                     *)
(* ============================================================================ *)

(* ------------------------------ *)
(* splitAt                        *)
(* ------------------------------ *)

(* [splitAt n xs] returns a tuple (xs1, xs2), with "append xs1 xs2 = xs" and  "length xs1 = n". If there are not
val splitAtAcc : ∀ α. LIST α → NAT → LIST α → (LIST α * LIST α)
let rec splitAtAcc revAcc n l =
  match l with
   | [] → (reverse revAcc, [])
   | x :: xs → if n ≤ 0 then (reverse revAcc, l) else splitAtAcc (x::revAcc) (n−1) xs
 end

val splitAt : ∀ α. NAT → LIST α → (LIST α * LIST α)
let splitAt n l =
   splitAtAcc [] n l
(*  match l with   | []    -> ([], [])   | x :: xs ->     if n <= 0 then ([], l) else     begin      let (l1, l2) = sp


declare termination_argument splitAt = automatic

declare isabelle target_rep function splitAt = 'split_at'
declare {ocaml, hol} rename function splitAt = split_at


assert splitAt_1 : (splitAt 0 [(1 : NAT); 2; 3; 4; 5; 6] = ([], [1; 2; 3; 4; 5; 6]))
assert splitAt_2 : (splitAt 2 [(1 : NAT); 2; 3; 4; 5; 6] = ([1; 2], [3; 4; 5; 6]))
assert splitAt_3 : (splitAt 100 [(1 : NAT); 2; 3; 4; 5; 6] = ([1; 2; 3; 4; 5; 6], []))
```

```
lemma splitAt_append : (∀ n xs.
  let (xs1, xs2) = splitAt n xs in
  (xs = xs1 ++ xs2))

lemma splitAt_length : (∀ n xs.
  let (xs1, xs2) = splitAt n xs in
  ((length xs1 = n) ∨
   ((length xs1 = length xs) ∧ null xs2)))


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* take                                                          *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* take n xs returns the prefix of xs of length n, or xs itself if n > length xs *)
val take : ∀ α. NAT → LIST α → LIST α
let take n l = fst (splitAt n l)

declare hol target_rep function take = 'TAKE'
declare isabelle target_rep function take = 'List.take'

assert take1 : (take 0 [(1 : NAT); 2; 3; 4; 5; 6] = [])
assert take2 : (take 2 [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2])
assert take3 : (take 100 [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6])

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* drop                                                          *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* [drop n xs] drops the first [n] elements of [xs]. It returns the empty list, if [n] > [length xs]. *)
val drop : ∀ α. NAT → LIST α → LIST α
let drop n l = snd (splitAt n l)

declare hol target_rep function drop = 'DROP'
declare isabelle target_rep function drop = 'List.drop'

assert drop1 : (drop 0 [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6])
assert drop2 : (drop 2 [(1 : NAT); 2; 3; 4; 5; 6] = [3; 4; 5; 6])
assert drop3 : (drop 100 [(1 : NAT); 2; 3; 4; 5; 6] = [])

lemma splitAt_take_drop : (∀ n xs. splitAt n xs = (take n xs, drop n xs))

let inline {hol} splitAt n xs = (take n xs, drop n xs)

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* splitWhile, takeWhile, and dropWhile                                                  *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val splitWhile_tr : ∀ α. (α → BOOL) → LIST α → LIST α → (LIST α * LIST α)
let rec splitWhile_tr p xs acc = match xs with
  | [] →
    (reverse acc, [])
  | x :: xs →
    if p x then
      splitWhile_tr p xs (x::acc)
    else
      (reverse acc, x::xs)
```

```
end

declare termination_argument splitWhile_tr = automatic

val splitWhile : ∀ α. (α → BOOL) → LIST α → (LIST α * LIST α)
let splitWhile p xs = splitWhile_tr p xs []

assert splitWhile_1 : (splitWhile ((>) 3) [(1 : NAT); 2; 3; 4; 5; 6] = ([1; 2], [3; 4; 5; 6]))
assert splitWhile_2 : (splitWhile ((≤) 6) ([] : LIST NAT) = ([], []))

(* [takeWhile p xs] takes the first elements of [xs] that satisfy [p]. *)
val takeWhile : ∀ α. (α → BOOL) → LIST α → LIST α
let takeWhile p l = fst (splitWhile p l)

(* [dropWhile p xs] drops the first elements of [xs] that satisfy [p]. *)
val dropWhile : ∀ α. (α → BOOL) → LIST α → LIST α
let dropWhile p l = snd (splitWhile p l)

assert dropWhile_0 : (dropWhile ((>) 3) [(1 : NAT); 2; 3; 4; 5; 6] = [3; 4; 5; 6])
assert dropWhile_1 : (dropWhile ((≥) 5) [(1 : NAT); 2; 3; 4; 5; 6] = [6])
assert dropWhile_2 : (dropWhile ((>) 100) [(1 : NAT); 2; 3; 4; 5; 6] = [])
assert dropWhile_3 : (dropWhile ((<) 10) [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6])

assert takeWhile_0 : (takeWhile ((>) 3) [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2])
assert takeWhile_1 : (takeWhile ((≥) 5) [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5])
assert takeWhile_2 : (takeWhile ((>) 100) [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6])
assert takeWhile_3 : (takeWhile ((<) 10) [(1 : NAT); 2; 3; 4; 5; 6] = [])

(* ---------------------------------------- *)
(* isPrefixOf                               *)
(* ---------------------------------------- *)

val isPrefixOf : ∀ α. Eq α ⇒ LIST α → LIST α → BOOL
let rec isPrefixOf l_1 l_2 = match (l_1, l_2) with
  | ([], _) → true
  | (_ :: _, []) → false
  | (x :: xs, y :: ys) → (x = y) ∧ isPrefixOf xs ys
end
declare termination_argument isPrefixOf = automatic

declare hol target_rep function isPrefixOf = 'isPREFIX'

assert isPrefixOf_0 : (isPrefixOf [] [(0 : NAT); 1; 2; 3; 4])
assert isPrefixOf_1 : (isPrefixOf [0] [(0 : NAT); 1; 2; 3; 4])
assert isPrefixOf_2 : (isPrefixOf [0; 1; 2] [(0 : NAT); 1; 2; 3; 4])
assert isPrefixOf_3 : ¬ (isPrefixOf [0; 2] [(0 : NAT); 1; 2; 3; 4])
assert isPrefixOf_4 : ¬ (isPrefixOf [(0 : NAT); 1; 2; 3; 4] [])

lemma isPrefixOf_alt_def : ∀ l_1 l_2. isPrefixOf l_1 l_2 ⟷ (∃ l_3. l_2 = (l_1 ++ l_3))
lemma isPrefixOf_sym : ∀ l. isPrefixOf l l
lemma isPrefixOf_trans : ∀ l_1 l_2 l_3. isPrefixOf l_1 l_2 ⟶ isPrefixOf l_2 l_3 ⟶ isPrefixOf l_1 l_3
lemma isPrefixOf_antisym : ∀ l_1 l_2. isPrefixOf l_1 l_2 ⟶ isPrefixOf l_2 l_1 ⟶ (l_1 = l_2)

(* ---------------------------------------- *)
(* update                                   *)
(* ---------------------------------------- *)
val update : ∀ α. LIST α → NAT → α → LIST α
let rec update l n e =
```

```
  match l with
   | [] → []
   | x :: xs → if n = 0 then e :: xs else x :: (update xs (n − 1) e)
end
declare termination_argument update = automatic

declare isabelle target_rep function update = 'List.list_update'
declare hol target_rep function update l n e = 'LUPDATE' e n l
declare {ocaml} rename function update = list_update

assert list_update_1 : (update [] 2 (3 : NAT) = [])
assert list_update_2 : (update [1;2;3;4;5] 0 (0 : NAT) = [0;2;3;4;5])
assert list_update_3 : (update [1;2;3;4;5] 1 (0 : NAT) = [1;0;3;4;5])
assert list_update_4 : (update [1;2;3;4;5] 2 (0 : NAT) = [1;2;0;4;5])
assert list_update_5 : (update [1;2;3;4;5] 5 (0 : NAT) = [1;2;3;4;5])

lemma list_update_length : (∀ l n e. length (update l n e) = length l)
lemma list_update_index : (∀ i l n e.
  (index (update l n e) i = ((if i = n ∧ n < length l then Just e else index l e))))



(* ========================================================================== *)
(* Searching lists                                                            *)
(* ========================================================================== *)

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* Membership test                                                            *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* The membership test, one of the basic list functions, is actually tricky for  Lem, because it is tricky, wh

val elem : ∀ α. Eq α ⇒ α → LIST α → BOOL
val elemBy : ∀ α. (α → α → BOOL) → α → LIST α → BOOL

let elemBy eq e l =  any (eq e) l
let elem =  elemBy (=)

declare hol target_rep function elem = 'MEM'
(* declare ocaml  target_rep function elem = `List.mem`  *)
declare isabelle target_rep function elem e l = 'Set.member' e ('set' l)

assert elem_1 : (elem (2 : NAT) [3;1;2;4])
assert elem_2 : (elem (3 : NAT) [3;1;2;4])
assert elem_3 : (elem (4 : NAT) [3;1;2;4])
assert elem_4 : (¬ (elem (5 : NAT) [3;1;2;4]))

lemma elem_spec : ((∀ e. ¬ (elem e [])) ∧
              (∀ e x xs. (elem e (x :: xs)) = ((e = x) ∨ (elem e xs))))

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* Find                                                                       *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
val find : ∀ α. (α → BOOL) → LIST α → MAYBE α  (* previously not of maybe type *)
let rec find P l =  match l with
  | [] → Nothing
  | x :: xs → if P x then Just x else find P xs
end
```

```
declare termination_argument find = automatic

declare isabelle target_rep function find = 'List.find'
declare {ocaml, hol} rename function find = list_find_opt

assert find_1 : ((find (fun n → n > (3 : NAT)) []) = Nothing)
assert find_2 : ((find (fun n → n > (3 : NAT)) [2; 1; 3]) = Nothing)
assert find_3 : ((find (fun n → n > (3 : NAT)) [2; 1; 5; 4]) = Just 5)
assert find_4 : ((find (fun n → n > (3 : NAT)) [2; 1; 4; 5; 4]) = Just 4)

lemma find_in : (∀ P l x. (find P l = Just x) ⟶ P x ∧ elem x l)
lemma find_not_in : (∀ P l. (find P l = Nothing) = (¬ (any P l)))


(* − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − *)
(* Lookup in an associative list *)
(* − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − *)
val lookup : ∀ α β. Eq α ⇒ α → LIST (α ∗ β) → MAYBE β
val lookupBy : ∀ α β. (α → α → BOOL) → α → LIST (α ∗ β) → MAYBE β

(* DPM : eta − expansion for Coq backend type − inference. *)
let lookupBy eq k m = Maybe.map (fun x → snd x) (find (fun (k′, _) → eq k k′) m)
let inline lookup = lookupBy (=)

declare isabelle target_rep function lookup x l = 'Map.map_of' l x
declare {ocaml, hol} rename function lookup = list_assoc_opt

assert lookup_1 : (lookup (3 : NAT) ([(4, (5 : NAT)); (3, 4); (1, 2); (3, 5)]) = Just 4)
assert lookup_2 : (lookup (8 : NAT) ([(4, (5 : NAT)); (3, 4); (1, 2); (3, 5)]) = Nothing)
assert lookup_3 : (lookup (1 : NAT) ([(4, (5 : NAT)); (3, 4); (1, 2); (3, 5)]) = Just 2)

(* − − − − − − − − − − − − − − − − − − − − − − − − − − − *)
(* filter *)
(* − − − − − − − − − − − − − − − − − − − − − − − − − − − *)
val filter : ∀ α. (α → BOOL) → LIST α → LIST α
let rec filter P l = match l with
                       | [] → []
                       | x :: xs → if (P x) then x :: (filter P xs) else filter P xs
                     end
declare termination_argument filter = automatic

declare hol target_rep function filter = 'FILTER'
declare ocaml target_rep function filter = 'List.filter'
declare isabelle target_rep function filter = 'List.filter'
declare coq target_rep function filter = 'List.filter'

assert filter_0 : (filter (fun x → x > (4 : NAT)) [] = [])
assert filter_1 : (filter (fun x → x > (4 : NAT)) [1; 2; 4; 5; 2; 7; 6] = [5; 7; 6])
lemma filter_nil_thm : (∀ P. filter P [] = [])
lemma filter_cons_thm : (∀ P x xs. filter P (x::xs) = (let l′ = filter P xs in (if (P x) then x :: l′ else l′)))


(* − − − − − − − − − − − − − − − − − − − − − − − − − − − *)
(* partition *)
(* − − − − − − − − − − − − − − − − − − − − − − − − − − − *)
val partition : ∀ α. (α → BOOL) → LIST α → LIST α ∗ LIST α
let partition P l = (filter P l, filter (fun x → ¬ (P x)) l)
```

```
val reversePartition : ∀ α. (α → BOOL) → LIST α → LIST α * LIST α
let reversePartition P l = partition P (reverse l)

let inline {hol} partition P l = reversePartition P (reverse l)
declare hol target_rep function reversePartition = 'PARTITION'
declare ocaml target_rep function partition = 'List.partition'
declare isabelle target_rep function partition = 'List.partition'

assert partition_0 : (partition (fun x → x > (4 : NAT)) [] = ([], []))
assert partition_1 : (partition (fun x → x > (4 : NAT)) [1;2;4;5;2;7;6] = ([5;7;6], [1;2;4;2]))
lemma partition_fst : (∀ P l. fst (partition P l) = filter P l)
lemma partition_snd : (∀ P l. snd (partition P l) = filter (fun x → ¬ (P x)) l)


(* ------------------------------------------------------------------------ *)
(* delete first element                                                     *)
(* with certain property                                                    *)
(* ------------------------------------------------------------------------ *)

val deleteFirst : ∀ α. (α → BOOL) → LIST α → MAYBE (LIST α)
let rec deleteFirst P l = match l with
                            | [] → Nothing
                          | x :: xs → if (P x) then Just xs else Maybe.map (fun xs' → x :: xs') (deleteFirst P xs)
                          end
declare termination_argument deleteFirst = automatic

declare isabelle target_rep function deleteFirst = 'delete_first'
declare {ocaml, hol} rename function deleteFirst = list_delete_first

assert deleteFirst_1 : (deleteFirst (fun x → x > (5 : NAT)) [3;6;7;1] = Just [3;7;1])
assert deleteFirst_2 : (deleteFirst (fun x → x > (15 : NAT)) [3;6;7;1] = Nothing)
assert deleteFirst_3 : (deleteFirst (fun x → x > (2 : NAT)) [3;6;7;1] = Just [6;7;1])


val delete : ∀ α. Eq α ⇒ α → LIST α → LIST α
val deleteBy : ∀ α. (α → α → BOOL) → α → LIST α → LIST α

let deleteBy eq x l = fromMaybe l (deleteFirst (eq x) l)
let inline delete = deleteBy (=)

declare isabelle target_rep function delete = 'remove'_1
declare {ocaml, hol} rename function delete = list_remove_1
declare {ocaml, hol} rename function deleteBy = list_delete

assert delete_1 : (delete (6 : NAT) [(3 : NAT);6;7;1] = [3;7;1])
assert delete_2 : (delete (4 : NAT) [(3 : NAT);6;7;1] = [3;6;7;1])
assert delete_3 : (delete (3 : NAT) [(3 : NAT);6;7;1] = [6;7;1])
assert delete_4 : (delete (3 : NAT) [(3 : NAT);3;6;7;1] = [3;6;7;1])


(* ======================================================================== *)
(* Zipping and unzipping lists                                              *)
(* ======================================================================== *)

(* ------------------------------------------------------------------------ *)
(* zip                                                                      *)
(* ------------------------------------------------------------------------ *)
```

```
(* zip takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements
val zip : ∀ α β. LIST α → LIST β → LIST (α * β) (* before combine *)
let rec zip l1 l2 = match (l1, l2) with
  | (x :: xs, y :: ys) → (x, y) :: zip xs ys
  | _ → []
end
declare termination_argument zip = automatic

declare isabelle target_rep function zip = 'List.zip'
declare {ocaml, hol} rename function zip = list_combine

assert zip1 : (zip [(1 : NAT); 2;3;4;5] [(2 : NAT); 3;4;5;6] = [(1, 2);(2, 3);(3, 4);(4, 5);(5, 6)])

(* this test rules out List.combine for ocaml and ZIP for HOL, but it's needed to make it a total function *)
assert zip2 : (zip [(1 : NAT); 2;3] [(2 : NAT); 3;4;5;6] = [(1, 2);(2, 3);(3, 4)])

(* ---------------------------------- *)
(* unzip                              *)
(* ---------------------------------- *)

val unzip : ∀ α β. LIST (α * β) → (LIST α * LIST β)
let rec unzip l = match l with
  | [] → ([], [])
  | (x, y) :: xys → let (xs, ys) = unzip xys in (x :: xs, y :: ys)
end
declare termination_argument unzip = automatic

declare hol target_rep function unzip = 'UNZIP'
declare isabelle target_rep function unzip = 'list_unzip'
declare ocaml target_rep function unzip = 'List.split'

assert unzip1 : (unzip ([] : LIST (NAT * NAT)) = ([], []))
assert unzip2 : (unzip [((1 : NAT), (2 : NAT));(2, 3);(3, 4)] = ([1;2;3], [2;3;4]))


instance ∀ α. SetType α ⇒ (SetType (LIST α))
 let setElemCompare = lexicographicCompareBy setElemCompare
end

(* ---------------------------------- *)
(* distinct elements                  *)
(* ---------------------------------- *)

val allDistinct : ∀ α. Eq α ⇒ LIST α → BOOL
let rec allDistinct l =
  match l with
   | [] → true
   | (x :: l') → ¬ (elem x l') ∧ allDistinct l'
 end
declare termination_argument allDistinct = automatic

declare hol target_rep function allDistinct = 'ALL_DISTINCT'

(* some more useful functions *)
val mapMaybe : ∀ α β. (α → MAYBE β) → LIST α → LIST β
let rec mapMaybe f xs =
  match xs with
 | [] → []
```

```
  | x :: xs  →
      match f x with
      | Nothing  →  mapMaybe f xs
      | Just y  →  y :: (mapMaybe f xs)
      end
  end

val mapi  :  ∀ α β. (NAT  →  α  →  β)  →  LIST α  →  LIST β
let rec mapiAux f (n  :  NAT) l =   match l with
  | []  →  []
  | x  ::  xs  →  (f n x) :: mapiAux f (n + 1) xs
end
let mapi f l =   mapiAux f 0 l

val deletes :  ∀ α. Eq α  ⇒  LIST α  →  LIST α  →  LIST α
let deletes xs ys =
   foldl (flip delete) xs ys

(* ========================================================================== *)
(* Comments (not clean yet, please ignore the rest of the file)            *)
(* ========================================================================== *)

(*  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* skipped from Haskell Lib*)
(*  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - intersperse :: a -> [a] -> [a]intercalate :: [a] -> [[a]] -


(*  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* skipped from Lem Lib    *)
(*  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - val for_all2 : forall 'a 'b. ('a -> 'b -> bool) -> list '

val catMaybes  :  ∀ α. LIST (MAYBE α)  →  LIST α
let rec catMaybes xs =
   match xs with
   | []  →
       []
   | (Nothing  ::  xs′)  →
       catMaybes xs′
   | (Just x  ::  xs′)  →
       x :: catMaybes xs′
  end
```

8 Either

```
(* ************************************************************************ *)
(* A library for sum types                                                  *)
(* ************************************************************************ *)

(* ======================================================================== *)
(* Header                                                                   *)
(* ======================================================================== *)

declare {isabelle, hol, coq} rename module = lem_either
declare {ocaml} rename module = Lem_either

open import Bool Basic_classes List Tuple
open import {hol} sumTheory
open import {ocaml} Either

type EITHER α β
  = LEFT of α
  | RIGHT of β

declare ocaml target_rep type EITHER = 'Either.either'
declare isabelle target_rep type EITHER = 'sum'
declare hol target_rep type EITHER = 'sum'
declare coq target_rep type EITHER = 'sum'

declare isabelle target_rep function Left = 'Inl'
declare isabelle target_rep function Right = 'Inr'
declare ocaml target_rep function Left = 'Either.Left'
declare ocaml target_rep function Right = 'Either.Right'
declare hol target_rep function Left = 'INL'
declare hol target_rep function Right = 'INR'
declare coq target_rep function Left = 'inl'
declare coq target_rep function Right = 'inr'


(* ------------------------------------------------------------------------ *)
(* Equality.                                                                *)
(* ------------------------------------------------------------------------ *)

val eitherEqual : ∀ α β. Eq α, Eq β ⇒ (EITHER α β) → (EITHER α β) → BOOL
val eitherEqualBy : ∀ α β. (α → α → BOOL) → (β → β → BOOL) → (EITHER α β) →
(EITHER α β) → BOOL

let eitherEqualBy eql eqr (left : EITHER α β) (right : EITHER α β) =
  match (left, right) with
    | (Left l, Left l') → eql l l'
    | (Right r, Right r') → eqr r r'
    | _ → false
  end
let eitherEqual = eitherEqualBy (=) (=)

let inline {hol, isabelle} eitherEqual = unsafe_structural_equality
let inline {ocaml} eitherEqual = eitherEqualBy (=) (=)
declare ocaml target_rep function eitherEqualBy = 'Either.eitherEqualBy'

instance ∀ α β. Eq α, Eq β ⇒ (Eq (EITHER α β))
  let = = eitherEqual
```

```
  let <> x y =  ¬ (eitherEqual x y)
 end

let either_setElemCompare cmpa cmpb x y =
   match (x,  y) with
    | (Left x′,  Left y′)  →  cmpa x′ y′
    | (Right x′,  Right y′)  →  cmpb x′ y′
    | (Left _,  Right _)  →  LT
    | (Right _,  Left _)  →  GT
  end

instance ∀ α β. SetType α,  SetType β  ⇒  (SetType (EITHER α β))
  let setElemCompare x y =  either_setElemCompare setElemCompare setElemCompare x y
end

assert either_equal1 :  (((Left false)  :  EITHER BOOL BOOL) = Left false)
assert either_equal2 :  (((Left true)  :  EITHER BOOL BOOL) ≠ Left false)
assert either_equal3 :  (((Left true)  :  EITHER BOOL BOOL) = Left true)
assert either_equal4 :  (((Right false)  :  EITHER BOOL BOOL) = Right false)
assert either_equal5 :  (((Right false)  :  EITHER BOOL BOOL) ≠ Right true)
assert either_equal6 :  (((Right true)  :  EITHER BOOL BOOL) ≠ Left true)
assert either_equal7 :  (((Left true)  :  EITHER BOOL BOOL) ≠ Right true)

assert either_pattern1 :  (match (Left true) with Left x  →  x | Right y  →  ¬ y end)
assert either_pattern2 :  (match (Right false) with Left x  →  x | Right y  →  ¬ y end)
assert either_pattern3 :  (¬ (match (Left false) with Left x  →  x | Right y  →  ¬ y end))
assert either_pattern4 :  (¬ (match (Right true) with Left x  →  x | Right y  →  ¬ y end))


(* ----------------------------------------------------------------------
(* Utility functions.                                        *)
(* ----------------------------------------------------------------------

val isLeft  :  ∀ α β. EITHER α β  →  BOOL
let inline isLeft  =  function
  | Left _  →  true
  | Right _  →  false
end

declare hol target_rep function isLeft  =  'ISL'

assert isLeft1  :  (isLeft ((Left true) :  EITHER BOOL BOOL))
assert isLeft2  :  (¬ (isLeft ((Right true) :  EITHER BOOL BOOL)))

val isRight  :  ∀ α β. EITHER α β  →  BOOL
let inline isRight  =  function
  | Right _  →  true
  | Left _  →  false
end

declare hol target_rep function isRight  =  'ISR'

assert isRight1  :  (isRight ((Right true) :  EITHER BOOL BOOL))
assert isRight2  :  (¬ (isRight ((Left true) :  EITHER BOOL BOOL)))


val either  :  ∀ α β γ. (α  →  γ)  →  (β  →  γ)  →  EITHER α β  →  γ
let either fa fb x =  match x with
```

```
  | Left a → fa a
  | Right b → fb b
end

declare ocaml target_rep function either = 'Either.either_case'
declare isabelle target_rep function either = 'case_sum'
declare hol target_rep function either fa fb x = 'sum_CASE' x fa fb

assert either_1 : (either ((fun b → ¬ b)) (fun b → b) (Left true) = false)
assert either_2 : (either ((fun b → ¬ b)) (fun b → b) (Left false) = true)
assert either_3 : (either ((fun b → ¬ b)) (fun b → b) (Right true) = true)
assert either_4 : (either ((fun b → ¬ b)) (fun b → b) (Right false) = false)


val partitionEither : ∀ α β. LIST (EITHER α β) → (LIST α * LIST β)
let rec partitionEither l = match l with
  | [] → ([], [])
  | x :: xs → begin
      let (ll, rl) = partitionEither xs in
      match x with
        | Left l → (l::ll, rl)
        | Right r → (ll, r::rl)
      end
    end
end
declare termination_argument partitionEither = automatic
declare {hol} rename function partitionEither = SUM_PARTITION

declare isabelle target_rep function partitionEither = 'sum_partition'
declare ocaml target_rep function partitionEither = 'Either.either_partition'

assert partitionEither_1 : (partitionEither [Left true; Right false; Right false; Left false; Right true] = ([true; false], [false; false;


val lefts : ∀ α β. LIST (EITHER α β) → LIST α
let inline lefts l = fst (partitionEither l)

assert lefts_1 : ((lefts [Left true; Right false; Right false; Left false; Right true]) = [true; false])


val rights : ∀ α β. LIST (EITHER α β) → LIST β
let inline rights l = snd (partitionEither l)


assert rights_1 : (rights [Left true; Right false; Right false; Left false; Right true] = [false; false; true])
```

9 Set_helpers

```
(* ************************************************************************** *)
(* Helper functions for sets                                                  *)
(* ************************************************************************** *)

(* Usually there is a something.lem file containing the main definitions and a  something_extra.lem one conta

(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)
```

open import *Bool Basic_classes Maybe Function Num*
declare {*isabelle*, *hol*, *ocaml*, *coq*} rename module = lem_set_helpers

open import {*coq*} *Coq.Lists.List*

```
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* fold                *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* fold is suspicious, because if given a function, for which  the order, in which the arguments are given, ma
```

val *fold* : $\forall\ \alpha\ \beta.\ (\alpha \to \beta \to \beta) \to \text{SET}\ \alpha \to \beta \to \beta$
declare compile_message fold = *"fold is non-deterministic because the order of the iteration is unclear. Its result may diffe*

declare *hol* target_rep function fold = `'ITSET'`
declare *isabelle* target_rep function fold $f\ A\ q$ = `'Finite_Set.fold'` $f\ q\ A$
declare *ocaml* target_rep function fold = `'Pset.fold'`
declare *coq* target_rep function fold = `'set_fold'`

10 Set

```
(* ************************************************************************ *)
(* A library for sets                                                       *)
(*                                                                          *)
(* It mainly follows the Haskell Set - library                              *)
(* ************************************************************************ *)

(* Sets in Lem are a bit tricky. On the one hand, we want efficiently executable sets.  OCaml and Haskell both


(* ========================================================================= *)
(* Header                                                                    *)
(* ========================================================================= *)

open import Bool Basic_classes Maybe Function Num List Set_helpers

declare {isabelle, hol, ocaml, coq} rename module = lem_set

(* DPM: sets currently implemented as lists due to mismatch between Coq type * class hierarchy and the hier
open import {coq} Coq.Lists.List
open import {hol} lemTheory
open import {isabelle} $LIB_DIR/Lem

(* Type of sets and set comprehensions are hard-coded *)

declare ocaml target_rep type SET = 'Pset.set'

(* ------------------------- *)
(* Equality check            *)
(* ------------------------- *)

val setEqualBy : ∀ α. (α → α → ORDERING) → SET α → SET α → BOOL
declare coq target_rep function setEqualBy = 'set_equal_by'

val setEqual : ∀ α. SetType α ⇒ SET α → SET α → BOOL
let inline {hol, isabelle} setEqual = unsafe_structural_equality
let inline {coq} setEqual = setEqualBy setElemCompare
declare ocaml target_rep function setEqual = 'Pset.equal'

instance ∀ α. SetType α ⇒ (Eq (SET α))
  let = =  setEqual
  let <> s1 s2 =  ¬ (setEqual s1 s2)
end

(* ------------------------- *)
(* Empty set                 *)
(* ------------------------- *)

val empty : ∀ α. SetType α ⇒ SET α
val emptyBy : ∀ α. (α → α → ORDERING) → SET α

declare ocaml target_rep function emptyBy = 'Pset.empty'
let inline {ocaml} empty = emptyBy setElemCompare

declare coq target_rep function empty = 'set_empty'
declare hol target_rep function empty = 'EMPTY'
declare isabelle target_rep function empty = '{}'
```

```
declare html target_rep function empty = '&empty;'
declare tex target_rep function empty = '$\emptyset$'

assert empty_0 : (∅ : set bool) = {}
assert empty_1 : (∅ : set nat) = {}
assert empty_2 : (∅ : set (list nat)) = {}
assert empty_3 : (∅ : set (set nat)) = {}

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* any / all                *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val any : ∀ α. SetType α ⇒ (α → bool) → set α → bool
let inline any P s = (∃ e ∈ s. P e)

declare coq target_rep function any = 'set_any'
declare hol target_rep function any P s = 'EXISTS' P ('SET_TO_LIST' s)
declare isabelle target_rep function any P s = 'Set.Bex' s P
declare ocaml target_rep function any = 'Pset.exists'

assert any_0 : any (fun (x : nat) → x > 5) {3, 4, 6}
assert any_1 : ¬ (any (fun (x : nat) → x > 10) {3, 4, 6})

val all : ∀ α. SetType α ⇒ (α → bool) → set α → bool
let inline all P s = (∀ e ∈ s. P e)

declare coq target_rep function all = 'set_for_all'
declare hol target_rep function all P s = 'EVERY' P ('SET_TO_LIST' s)
declare isabelle target_rep function all P s = 'Set.Ball' s P
declare ocaml target_rep function all = 'Pset.for_all'

assert all_0 : all (fun (x : nat) → x > 2) {3, 4, 6}
assert all_1 : ¬ (all (fun (x : nat) → x > 2) {3, 4, 6, 1})


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* (IN)                     *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val IN [member] : ∀ α. SetType α ⇒ α → set α → bool
val memberBy : ∀ α. (α → α → ordering) → α → set α → bool

declare coq target_rep function memberBy = 'set_member_by'
let inline {coq} member = memberBy setElemCompare
declare ocaml target_rep function member = 'Pset.mem'
declare isabelle target_rep function member = infix '\<in>'
declare hol target_rep function member = infix 'IN'
declare html target_rep function member = infix '&isin;'
declare tex target_rep function member = infix '$\in$'

assert in_1 : ((1 : nat) ∈ {(2 : nat), 3, 1})
assert in_2 : (¬ ((1 : nat) ∈ {2, 3, 4}))
assert in_3 : (¬ ((1 : nat) ∈ {}))
assert in_4 : ((1 : nat) ∈ {1, 2, 1, 3, 1, 4})

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* not (IN)                 *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
```

```
val NIN [notMember] : ∀ α. SetType α ⇒ α → SET α → BOOL
let inline notMember e s = ¬ (e ∈ s)
declare html target_rep function notMember = infix '&notin;'
declare isabelle target_rep function notMember = infix '\<notin>'
declare tex target_rep function notMember = infix '$\not\in$'

assert nin_1 : ¬ ((1 : NAT) ∉ {2, 3, 1})
assert nin_2 : ((1 : NAT) ∉ {2, 3, 4})
assert nin_3 : ((1 : NAT) ∉ {})
assert nin_4 : ¬ ((1 : NAT) ∉ {1, 2, 1, 3, 1, 4})



(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* Emptyness check       *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val null : ∀ α. SetType α ⇒ SET α → BOOL (* before is_empty *)
let inline null s = (s = {})

declare ocaml target_rep function null = 'Pset.is_empty'
declare coq target_rep function null = 'set_is_empty'

assert null_1 : (null ({} : SET NAT))
assert null_2 : (¬ (null {(1 : NAT)}))


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* singleton             *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val singletonBy : ∀ α. (α → α → ORDERING) → α → SET α
val singleton : ∀ α. SetType α ⇒ α → SET α

declare ocaml target_rep function singletonBy = 'Pset.singleton'
declare coq target_rep function singleton = 'set_singleton'

let inline {ocaml} singleton = singletonBy setElemCompare
let inline ~{ocaml, coq} singleton x = {x}

assert singleton_1 : singleton (2 : NAT) = {2}
assert singleton_2 : ¬ (null (singleton (2 : NAT)))
assert singleton_3 : 2 ∈ (singleton (2 : NAT))
assert singleton_4 : 3 ∉ (singleton (2 : NAT))


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* size                  *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val size : ∀ α. SetType α ⇒ SET α → NAT

declare ocaml target_rep function size = 'Pset.cardinal'
declare coq target_rep function size = 'set_cardinal'
declare hol target_rep function size = 'CARD'
declare isabelle target_rep function size = 'card'
```

```
assert size_1 : (size ({} : SET NAT) = 0)
assert size_2 : (size {(2 : NAT)} = 1)
assert size_3 : (size {(1 : NAT), 1} = 1)
assert size_4 : (size {(2 : NAT), 1, 3} = 3)
assert size_5 : (size {(2 : NAT), 1, 3, 9} = 4)

lemma null_size : (∀ s. (null s) ⟶ (size s = 0))
lemma null_singleton : (∀ x. (size (singleton x) = 1))


(* ----------------------------------------------------------------------*)
(* setting up pattern matching *)
(* ---------------------------------------------------------------------- *)

val set_case : ∀ α β. SetType α ⇒ SET α → β → (α → β) → β → β

(* please provide target bindings, since choose is defined only in extra  and not the right thing to use here

declare hol target_rep function set_case = 'set_CASE'
declare isabelle target_rep function set_case = 'set_case'
declare coq target_rep function set_case = 'set_case'
declare ocaml target_rep function set_case = 'Pset.set_case'

declare pattern_match inexhaustive SET α = [ empty; singleton ] set_case

assert set_patterns_0 : (
  match ({} : SET NAT) with
    | ∅ → true
    | _ → false
  end
)

assert set_patterns_1 : ¬ (
  match {(2 : NAT)} with
    | ∅ → true
    | _ → false
  end
)

assert set_patterns_2 : ¬ (
  match {(3 : NAT), 4} with
    | ∅ → true
    | _ → false
  end
)

assert set_patterns_3 : (
  match ({2} : SET NAT) with
    | ∅ → 0
    | singleton x → x
    | _ → 1
  end
) = 2

assert set_patterns_4 : (
  match ({} : SET NAT) with
    | ∅ → 0
    | singleton x → x
```

```
    | _ → 1
  end
) = 0

assert set_patterns5 : (
  match ({3, 4, 5} : SET NAT) with
    | ∅ → 0
    | singleton x → x
    | _ → 1
  end
) = 1

assert set_patterns6 : (
  match ({3, 3, 3} : SET NAT) with
    | ∅ → 0
    | singleton x → x
    | _ → 1
  end
) = 3

assert set_patterns7 : (
  match ({3, 4, 5} : SET NAT) with
    | ∅ → 0
    | singleton _ → 1
    | s → size s
  end
) = 3

assert set_patterns8 : (
  match (({3, 4, 5} : SET NAT), false) with
    | (∅, true) → 0
    | (singleton _, _) → 1
    | (s, true) → size s
    | _ → 5
  end
) = 5

assert set_patterns9 : (
  match ({5} : SET NAT) with
    | ∅ → 0
    | singleton 2 → 0
    | singleton (x + 3) → x
    | _ → 1
  end
) = 2

assert set_patterns10 : (
  match ({2} : SET NAT) with
    | ∅ → 0
    | singleton 2 → 0
    | singleton (x + 3) → x
    | _ → 1
  end
) = 0


(* -------------------------- *)
(* union                      *)
```

```
(* -------------------------- *)

val unionBy : ∀ α. (α → α → ORDERING) → SET α → SET α → SET α
val union : ∀ α. SetType α ⇒ SET α → SET α → SET α
declare ocaml target_rep function union = 'Pset.(union)'
declare hol target_rep function union = infix 'UNION'
declare isabelle target_rep function union = infix '\<union>'
declare coq target_rep function unionBy = 'set_union_by'
declare tex target_rep function union = infix '$\cup$'
let inline {coq} union = unionBy setElemCompare


assert union1 : ({(1 : NAT), 2, 3} ∪ {3, 2, 4} = {1, 2, 3, 4})
lemma union_in : (∀ e s1 s2. e ∈ (s1 ∪ s2) ⟷ (e ∈ s1 ∨ e ∈ s2))

(* ------------------------- *)
(* insert            *)
(* ------------------------- *)

val insert : ∀ α. SetType α ⇒ α → SET α → SET α (* before add *)

declare ocaml target_rep function insert = 'Pset.add'
declare coq target_rep function insert = 'set_add'
declare hol target_rep function insert = infix 'INSERT'
declare isabelle target_rep function insert = 'Set.insert'

assert insert1 : ((insert (2 : NAT) {3, 4}) = {2, 3, 4})
assert insert2 : ((insert (3 : NAT) {3, 4}) = {3, 4})
assert insert3 : ((insert (3 : NAT) {}) = {3})

(* ------------------------- *)
(* filter            *)
(* ------------------------- *)

val filter : ∀ α. SetType α ⇒ (α → BOOL) → SET α → SET α
let filter P s = {e | ∀ e ∈ s | P e}

declare ocaml target_rep function filter = 'Pset.filter'
declare isabelle target_rep function filter = 'set_filter'
declare hol target_rep function filter = 'SET_FILTER'

assert filter1 : (filter (fun n → (n > 2)) {(1 : NAT), 2, 3, 4} = {3, 4})
assert filter2 : (filter (fun n → n > (2 : NAT)) {} = {})
lemma filter_emp : (∀ P. (filter P {}) = {})
lemma filter_insert : (∀ e s P. (filter P (insert e s)) =
  (if (P e) then insert e (filter P s) else (filter P s)))


(* ------------------------- *)
(* partition         *)
(* ------------------------- *)

val partition : ∀ α. SetType α ⇒ (α → BOOL) → SET α → SET α * SET α
let partition P s = (filter P s, filter (fun e → ¬ (P e)) s)
declare {hol} rename function partition = SET_PARTITION


(* ------------------------- *)
```

```
(* split                *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val split : ∀ α. SetType α, Ord α ⇒ α → set α → set α * set α
let split p s = (filter ((>) p) s, filter ((<) p) s)
declare {hol} rename function split = SET_SPLIT

val splitMember : ∀ α. SetType α, Ord α ⇒ α → set α → set α * bool * set α
let splitMember p s = (filter ((<) p) s, p ∈ s, filter ((>) p) s)

assert split_simple : split
      (3, 0)
      ({ (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0) } : set ($\mathbb{N}$ * $\mathbb{N}$))
      = ({ (1, 0), (2, 0) }, { (4, 0), (5, 0), (6, 0) })

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* subset and proper subset *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val isSubsetOfBy : ∀ α. (α → α → ordering) → set α → set α → bool
val isProperSubsetOfBy : ∀ α. (α → α → ordering) → set α → set α → bool

val isSubsetOf : ∀ α. SetType α ⇒ set α → set α → bool
val isProperSubsetOf : ∀ α. SetType α ⇒ set α → set α → bool

declare ocaml target_rep function isSubsetOf = 'Pset.subset'
declare hol target_rep function isSubsetOf = infix 'SUBSET'
declare isabelle target_rep function isSubsetOf = infix '\<subseteq>'
declare html target_rep function isSubsetOf = infix '&sube;'
declare tex target_rep function isSubsetOf = infix '$\subseteq$'
declare coq target_rep function isSubsetOfBy = 'set_subset_by'
let inline {coq} isSubsetOf = isSubsetOfBy setElemCompare

declare ocaml target_rep function isProperSubsetOf = 'Pset.subset_proper'
declare hol target_rep function isProperSubsetOf = infix 'PSUBSET'
declare isabelle target_rep function isProperSubsetOf = infix '\<subset>'
declare html target_rep function isProperSubsetOf = infix '&sub;'
declare tex target_rep function isProperSubsetOf = infix '$\subset$'
declare coq target_rep function isProperSubsetOfBy = 'set_proper_subset_by'
let inline {coq} isProperSubsetOf = isProperSubsetOfBy setElemCompare

let inline subset = (⊆)
declare tex target_rep function subset = infix '$\subseteq$'

assert isSubsetOf_1 : (({} : set nat) ⊆ {})
assert isSubsetOf_2 : ({(1 : nat), 2, 3} ⊆ {1, 2, 3})
assert isSubsetOf_3 : ({(1 : nat), 2} ⊆ {3, 2, 1})
lemma isSubsetOf_refl : (∀ s. s ⊆ s)
lemma isSubsetOf_def : (∀ s1 s2. s1 ⊆ s2 = (∀ e. e ∈ s1 ⟶ e ∈ s2))
lemma isSubsetOf_eq : (∀ s1 s2. (s1 = s2) ⟷ ((s1 ⊆ s2) ∧ (s2 ⊆ s1)))

assert isProperSubsetOf_1 : (¬ (({} : set nat) ⊂ {}))
assert isProperSubsetOf_2 : (¬ ({(1 : nat), 2, 3} ⊂ {1, 2, 3}))
assert isProperSubsetOf_3 : ({(1 : nat), 2} ⊂ {3, 2, 1})
lemma isProperSubsetOf_irrefl : (∀ s. ¬ (s ⊂ s))
lemma isProperSubsetOf_def : (∀ s1 s2. s1 ⊂ s2 ⟷ ((s1 ⊆ s2) ∧ ¬ (s2 ⊆ s1)))
```

```
(* ------------------------- *)
(* delete                    *)
(* ------------------------- *)

val delete : ∀ α. SetType α, Eq α ⇒ α → SET α → SET α
val deleteBy : ∀ α. SetType α ⇒ (α → α → BOOL) → α → SET α → SET α

let inline deleteBy eq e s = filter (fun e2 → ¬ (eq e e2)) s
let inline delete e s = deleteBy (=) e s


(* ------------------------- *)
(* bigunion                  *)
(* ------------------------- *)

val bigunion : ∀ α. SetType α ⇒ SET (SET α) → SET α
val bigunionBy : ∀ α. (α → α → ORDERING) → SET (SET α) → SET α

let bigunion bs = {x | ∀ s ∈ bs x ∈ s | true}

declare ocaml target_rep function bigunionBy = 'Pset.bigunion'
let inline {ocaml} bigunion = bigunionBy setElemCompare
declare hol target_rep function bigunion = 'BIGUNION'
declare isabelle target_rep function bigunion = '\<Union>'
declare tex target_rep function bigunion = '$\bigcup$'

assert bigunion0 : (⋃ {{(1 : NAT)}} = {1})
assert bigunion1 : (⋃ {{(1 : NAT), 2, 3} , {3, 2, 4}} = {1, 2, 3, 4})
assert bigunion2 : (⋃ {{(1 : NAT), 2, 3} , {3, 2, 4}, {}} = {1, 2, 3, 4})
assert bigunion3 : (⋃ {{(1 : NAT), 2, 3} , {3, 2, 4}, {5}} = {1, 2, 3, 4, 5})
lemma bigunion_in : (∀ e bs. e ∈ ⋃ bs ⟷ (∃ s. s ∈ bs ∧ e ∈ s))

(* ------------------------- *)
(* big intersection          *)
(* ------------------------- *)

(* Shaked's addition, for which he is now forever responsible as a de facto * Lem maintainer... *)
val bigintersection : ∀ α. SetType α ⇒ SET (SET α) → SET α
let bigintersection bs = {x | ∀ x ∈ (⋃ bs) | ∀ s ∈ bs. x ∈ s}

(* ------------------------- *)
(* difference                *)
(* ------------------------- *)

val differenceBy : ∀ α. (α → α → ORDERING) → SET α → SET α → SET α
val difference : ∀ α. SetType α ⇒ SET α → SET α → SET α
declare ocaml target_rep function difference = 'Pset.diff'
declare hol target_rep function difference = infix 'DIFF'
declare isabelle target_rep function difference = infix '-'
declare tex target_rep function difference = infix '$\setminus$'
declare coq target_rep function differenceBy = 'set_diff_by'
let inline {coq} difference = differenceBy setElemCompare

let inline \ = (\)

assert difference1 : ({(1 : NAT), 2, 3} \ {3, 2, 4} = {1})
```

lemma *difference_in* : $(\forall\ e\ s_1\ s_2.\ e \in (s_1 \setminus s_2) \longleftrightarrow (e \in s_1 \wedge \neg\ (e \in s_2)))$

```
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* intersection          *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
```

val *intersection* : $\forall\ \alpha.$ *SetType* $\alpha\ \Rightarrow$ SET $\alpha\ \rightarrow$ SET $\alpha\ \rightarrow$ SET α
val *intersectionBy* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow$ ORDERING$)\ \rightarrow$ SET $\alpha\ \rightarrow$ SET $\alpha\ \rightarrow$ SET α

declare *ocaml* target_rep function intersection = `'Pset.inter'`
declare *hol* target_rep function intersection = infix `'INTER'`
declare *isabelle* target_rep function intersection = infix `'\<inter>'`
declare *coq* target_rep function intersectionBy = `'set_inter_by'`
declare *tex* target_rep function intersection = infix `'$\cap$'`
let inline {*coq*} *intersection* = intersectionBy setElemCompare
let inline *inter* = $(\cap)$
declare *tex* target_rep function inter = infix `'$\cap$'`

assert $intersection_1$: $(\{1, 2, 3\} \cap \{(3 : \text{NAT}), 2, 4\} = \{2, 3\})$
lemma *intersection_in* : $(\forall\ e\ s_1\ s_2.\ e \in (s_1 \cap s_2) \longleftrightarrow (e \in s_1 \wedge e \in s_2))$

```
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* map                   *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
```

val *map* : $\forall\ \alpha\ \beta.$ *SetType* α, *SetType* $\beta\ \Rightarrow\ (\alpha\ \rightarrow\ \beta)\ \rightarrow$ SET $\alpha\ \rightarrow$ SET β (* before image *)
let *map* $f\ s =$ $\{\ f\ e \mid \forall\ e \in s \mid$ true $\}$

val *mapBy* : $\forall\ \alpha\ \beta.\ (\beta\ \rightarrow\ \beta\ \rightarrow$ ORDERING$)\ \rightarrow\ (\alpha\ \rightarrow\ \beta)\ \rightarrow$ SET $\alpha\ \rightarrow$ SET β

declare *ocaml* target_rep function mapBy = `'Pset.map'`

let inline {*ocaml*} *map* = mapBy setElemCompare
declare *hol* target_rep function map = `'IMAGE'`
declare *isabelle* target_rep function map = `'Set.image'`

assert map_1 : (map succ $\{(2 : \text{NAT}), 3, 4\} = \{5, 4, 3\}$)
assert map_2 : (map (fun $n\ \rightarrow\ n * 3$) $\{(2 : \text{NAT}), 3, 4\} = \{6, 9, 12\}$)

```
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* bigunionMap           *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* In order to avoid providing an comparison function for sets of sets,  it might be better to combine biguni
```

val *bigunionMap* : $\forall\ \alpha\ \beta.$ *SetType* α, *SetType* $\beta\ \Rightarrow\ (\alpha\ \rightarrow$ SET $\beta)\ \rightarrow$ SET $\alpha\ \rightarrow$ SET β
val *bigunionMapBy* : $\forall\ \alpha\ \beta.\ (\beta\ \rightarrow\ \beta\ \rightarrow$ ORDERING$)\ \rightarrow\ (\alpha\ \rightarrow$ SET $\beta)\ \rightarrow$ SET $\alpha\ \rightarrow$ SET β

let inline *bigunionMap* $f\ bs\ =\ \bigcup$ (map $f\ bs$)

declare *ocaml* target_rep function bigunionMapBy = `'Pset.map_union'`
let inline {*ocaml*} *bigunionMap* = bigunionMapBy setElemCompare

assert $bigunionmap_0$: (bigunionMap (fun $n\ \rightarrow\ \{n,\ 2 * n,\ 3 * n\}$) $\{(1 : \text{NAT})\} = \{1, 2, 3\}$)
assert $bigunionmap_1$: (bigunionMap (fun $n\ \rightarrow\ \{n,\ 2 * n,\ 3 * n\}$) $\{(2 : \text{NAT}), 8\} = \{2, 4, 6, 8, 16, 24\}$)

```
(* ---------------------------------------- *)
(* mapMaybe and fromMaybe                   *)
(* ---------------------------------------- *)

(* If the mapping function returns Just x, x is added to the result  set. If it returns Nothing, no element is

val mapMaybe : ∀ α β. SetType α, SetType β ⇒ (α → MAYBE β) → SET α → SET β
let setMapMaybe f s =
  bigunionMap (fun x → match f x with
                | Just y → singleton y
                | Nothing → ∅
                end)
        s

(* The name mapMaybe is already being used in the list.lem *)
declare rename function mapMaybe = setMapMaybe

val removeMaybe : ∀ α. SetType α ⇒ SET (MAYBE α) → SET α
let removeMaybe s =  setMapMaybe (fun x → x) s

(* ---------------------------------------- *)
(* min and max                              *)
(* ---------------------------------------- *)

val findMin : ∀ α. SetType α, Eq α ⇒ SET α → MAYBE α
val findMax : ∀ α. SetType α, Eq α ⇒ SET α → MAYBE α

(* Informal, since THE is not supported by all backendsval findMinBy : forall 'a. ('a -> 'a -> bool) -> (

declare ocaml target_rep function findMin = 'Pset.min_elt_opt'
declare ocaml target_rep function findMax = 'Pset.max_elt_opt'

(* XXX: move into Lem libraries... *)
declare isabelle target_rep function findMin = 'Elf_Types_Local.find_min_element'
declare isabelle target_rep function findMax = 'Elf_Types_Local.find_max_element'

declare hol target_rep function findMin = 'ARB'
declare hol target_rep function findMax = 'ARB'

(* ---------------------------------------- *)
(* fromList                                 *)
(* ---------------------------------------- *)

val fromList : ∀ α. SetType α ⇒ LIST α → SET α  (* before from_list *)
val fromListBy : ∀ α. (α → α → ORDERING) → LIST α → SET α

declare ocaml target_rep function fromListBy = 'Pset.from_list'
let inline {ocaml} fromList = fromListBy setElemCompare
declare hol target_rep function fromList = 'LIST_TO_SET'
declare isabelle target_rep function fromList = 'List.set'
declare coq target_rep function fromListBy = 'set_from_list_by'
let inline {coq} fromList = fromListBy setElemCompare


assert fromList_1 : (fromList [(2 : NAT); 4; 3] = {2, 3, 4})
assert fromList_2 : (fromList [(2 : NAT); 2; 3; 2; 4] = {2, 3, 4})
assert fromList_3 : (fromList ([] : LIST NAT) = {})
```

```
(* -------------------------- *)
(* Sigma                  *)
(* -------------------------- *)

val sigma  : ∀ α β. SetType α, SetType β ⇒ SET α → (α → SET β) → SET (α * β)
val sigmaBy : ∀ α β. ((α * β) → (α * β) → ORDERING) → SET α → (α → SET β) → SET (α * β)

declare ocaml target_rep function sigmaBy = 'Pset.sigma'

let sigma sa sb =  { (a, b) | ∀ a ∈ sa b ∈ sb a | true }
let inline {ocaml} sigma = sigmaBy setElemCompare

declare isabelle target_rep function sigma = 'Sigma'
declare coq target_rep function sigmaBy = 'set_sigma_by'
let inline {coq} sigma = sigmaBy setElemCompare
declare hol target_rep function sigma = 'SET_SIGMA'

assert Sigma_1 : (sigma {(2 : NAT), 3} (fun n → {n*2, n * 3}) = {(2, 4), (2, 6), (3, 6), (3, 9)})
lemma Sigma_2 : (∀ sa sb a b. ((a, b) ∈ sigma sa sb) ⟷ ((a ∈ sa) ∧ (b ∈ sb a)))


(* -------------------------- *)
(* cross product          *)
(* -------------------------- *)

val cross  : ∀ α β. SetType α, SetType β ⇒ SET α → SET β → SET (α * β)
val crossBy : ∀ α β. ((α * β) → (α * β) → ORDERING) → SET α → SET β → SET (α * β)

declare ocaml target_rep function crossBy = 'Pset.cross'

let cross s_1 s_2 =  { (e_1, e_2) | ∀ e_1 ∈ s_1 e_2 ∈ s_2 | true }

declare isabelle target_rep function cross = infix '\<times>'
declare hol target_rep function cross = infix 'CROSS'
declare tex target_rep function cross = infix '$\times$'
let inline {ocaml} cross = crossBy setElemCompare

lemma cross_by_sigma : ∀ s_1 s_2. s_1 × s_2 = sigma s_1 (const s_2)
assert cross_1 : ({(2 : NAT), 3} × {true, false} = {(2, true), (3, true), (2, false), (3, false)})


(* -------------------------- *)
(* finite                 *)
(* -------------------------- *)

val finite : ∀ α. SetType α ⇒ SET α → BOOL

let inline {ocaml, coq} finite _s = true
declare hol target_rep function finite = 'FINITE'
declare isabelle target_rep function finite = 'finite'


(* ------------------------------ *)
(* fixed point              *)
(* ----------------------------- *)

val leastFixedPoint : ∀ α. SetType α
```

```
  ⇒ NAT → (SET α → SET α) → SET α → SET α
let rec leastFixedPoint bound f x =
  match bound with
 | 0 → x
 | bound' + 1 → let fx = f x in
                if fx ⊆ x then x
                else leastFixedPoint bound' f (fx ∪ x)
 end

declare {isabelle} termination_argument leastFixedPoint = automatic

assert lfp_empty_0 : leastFixedPoint 0 (map (fun x → x)) ({} : SET NAT) = {}
assert lfp_empty_1 : leastFixedPoint 1 (map (fun x → x)) ({} : SET NAT) = {}
assert lfp_saturate_neg_1 : leastFixedPoint 1 (map (fun x → −x)) ({1, 2, 3} : SET INT) = {−3, −2, −1, 1, 2, 3}

assert lfp_saturate_neg_2 : leastFixedPoint 2 (map (fun x → −x)) ({1, 2, 3} : SET INT) = {−3, −2, −1, 1, 2, 3}

assert lfp_saturate_mod_3 : leastFixedPoint 3 (map (fun x → (2∗x) mod 5)) ({1} : SET NAT) = {1, 2, 3, 4}

assert lfp_saturate_mod_4 : leastFixedPoint 4 (map (fun x → (2∗x) mod 5)) ({1} : SET NAT) = {1, 2, 3, 4}

assert lfp_saturate_mod_5 : leastFixedPoint 5 (map (fun x → (2∗x) mod 5)) ({1} : SET NAT) = {1, 2, 3, 4}

assert lfp_termination : {1, 3, 5, 7, 9} ⊆ leastFixedPoint 5 (map (fun x → 2+x)) {(1 : $\MATHBB{N}$)}
```

11 Map

```
(* ************************************************************************ *)
(* A library for finite maps                                                *)
(* ************************************************************************ *)

(* ========================================================================= *)
(* Header                                                                    *)
(* ========================================================================= *)

declare {isabelle, ocaml, hol, coq} rename module = lem_map

open import Bool Basic_classes Function Maybe List Tuple Set Num
open import {hol} finite_mapTheory finite_mapLib

type MAP 'k 'v
declare ocaml target_rep type MAP = 'Pmap.map'
declare isabelle target_rep type MAP = 'Map.map'
declare hol target_rep type MAP = 'fmap'
declare coq target_rep type MAP = 'fmap'



(* ------------------------------------------------------------------------- *)
(* Map equality.                                                             *)
(* ------------------------------------------------------------------------- *)

val mapEqual : ∀ 'k 'v. Eq 'k, Eq 'v ⇒ MAP 'k 'v → MAP 'k 'v → BOOL
val mapEqualBy : ∀ 'k 'v. ('k → 'k → BOOL) → ('v → 'v → BOOL) → MAP 'k 'v → MAP 'k 'v → BOOL


declare ocaml target_rep function mapEqualBy eq_k eq_v = 'Pmap.equal' eq_v
declare coq target_rep function mapEqualBy = 'fmap_equal_by'
let inline ~{hol, isabelle} mapEqual = mapEqualBy (=) (=)
let inline {hol, isabelle} mapEqual = unsafe_structural_equality

instance ∀ 'k 'v. Eq 'k, Eq 'v ⇒ (Eq (MAP 'k 'v))
  let = = mapEqual
  let <> m1 m2 = ¬ (mapEqual m1 m2)
end



(* ------------------------------------------------------------------------- *)
(* Map type class                                                            *)
(* ------------------------------------------------------------------------- *)

class ( MapKeyType α )
  val {ocaml, coq} mapKeyCompare : α → α → ORDERING
end

default_instance ∀ α. SetType α ⇒ ( MapKeyType α )
  let mapKeyCompare = setElemCompare
 end

(* ------------------------------------------------------------------------- *)
(* Empty maps                                                                *)
(* ------------------------------------------------------------------------- *)
```

```
val empty : ∀ 'k 'v. MapKeyType 'k ⇒ MAP 'k 'v
val emptyBy : ∀ 'k 'v. ('k → 'k → ORDERING) → MAP 'k 'v

declare ocaml target_rep function emptyBy = 'Pmap.empty'

let inline {ocaml} empty = emptyBy mapKeyCompare
declare coq target_rep function empty = 'fmap_empty'
declare hol target_rep function empty = 'FEMPTY'
declare isabelle target_rep function empty = 'Map.empty'


(* -------------------------------------------------------
(* Insertion                                    *)
(* -------------------------------------------------------

val insert : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → 'v → MAP 'k 'v → MAP 'k 'v

declare coq target_rep function insert = 'fmap_add'
declare ocaml target_rep function insert = 'Pmap.add'
(* declare hol     target_rep function insert k v m = `FUPDATE` m (k, v) *)
declare hol target_rep function insert k v m = special "%e |+ (%e, %e)" m k v

declare isabelle target_rep function insert = 'map_update'


(* -------------------------------------------------------
(* Singleton                                    *)
(* -------------------------------------------------------

val singleton : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → 'v → MAP 'k 'v
let inline singleton k v = insert k v empty

assert insert_equal_singleton : (mapEqual (insert (42 : NAT) false empty)
                                  (singleton 42 false))
assert commutative_insert_1 : (mapEqual
                        (insert (8 : NAT) true (insert 5 false empty))
                        (insert 5 false (insert 8 true empty)))
assert commutative_insert_2 : (¬ (mapEqual
                        (insert (8 : NAT) true (insert 8 false empty))
                        (insert 8 false (insert 8 true empty))))



(* -------------------------------------------------------
(* Emptyness check                                *)
(* -------------------------------------------------------

val null : ∀ 'k 'v. MapKeyType 'k, Eq 'k, Eq 'v ⇒ MAP 'k 'v → BOOL
let inline null m = (m = empty)

declare coq target_rep function null = 'fmap_is_empty'
declare ocaml target_rep function null = 'Pmap.is_empty'

assert empty_null : (null (empty : MAP NAT BOOL))


(* -------------------------------------------------------
(* lookup                                       *)
```

```
(* ---------------------------------------------------------------

val lookupBy : ∀ 'k 'v. ('k → 'k → ORDERING) → 'k → MAP 'k 'v → MAYBE 'v
declare coq target_rep function lookupBy = 'fmap_lookup_by'

val lookup : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → MAP 'k 'v → MAYBE 'v
let inline {coq} lookup = lookupBy mapKeyCompare
declare isabelle target_rep function lookup k m = ''m k
declare hol target_rep function lookup k m = 'FLOOKUP' m k
declare ocaml target_rep function lookup = 'Pmap.lookup'

assert lookup_insert_1 : (lookup 16 (insert (16 : NAT) true empty) = Just true)
assert lookup_insert_2 : (lookup 16 (insert 36 false (insert (16 : NAT) true empty)) = Just true )
assert lookup_insert_3 : (lookup 36 (insert 36 false (insert (16 : NAT) true empty)) = Just false )

assert lookup_empty_0 : (lookup 25 (empty : MAP NAT BOOL) = Nothing)
assert find_insert_0 : (lookup 16 (insert (16 : NAT) true empty) = Just true)

lemma lookup_empty : (∀ k. lookup k empty = Nothing)
lemma lookup_insert : (∀ k k' v m. lookup k (insert k' v m) = (if (k = k') then Just v else lookup k m))

(* ---------------------------------------------------------------
(* findWithDefault                                              *)
(* ---------------------------------------------------------------

val findWithDefault : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → 'v → MAP 'k 'v → 'v
let inline findWithDefault k v m = fromMaybe v (lookup k m)

(* ---------------------------------------------------------------
(* from lists                                                   *)
(* ---------------------------------------------------------------

val fromList : ∀ 'k 'v. MapKeyType 'k ⇒ LIST ('k * 'v) → MAP 'k 'v
let fromList l = foldl (fun m (k, v) → insert k v m) empty l

declare isabelle target_rep function fromList l = 'Map.map_of' (reverse l)
declare hol target_rep function fromList l = 'FUPDATE_LIST' 'FEMPTY' l

assert fromList_0 : (fromList [((2 : NAT), true); ((3 : NAT), true); ((4 : NAT), false)] =
                fromList [((4 : NAT), false); ((3 : NAT), true); ((2 : NAT), true)])
(* later entries have priority *)
assert fromList_1 : (fromList [((2 : NAT), true); ((2 : NAT), false); ((3 : NAT), true); ((4 : NAT), false)] =
                fromList [((4 : NAT), false); ((3 : NAT), true); ((2 : NAT), false)])


(* ---------------------------------------------------------------
(* to sets / domain / range                                     *)
(* ---------------------------------------------------------------

val toSet : ∀ 'k 'v. MapKeyType 'k, SetType 'k, SetType 'v ⇒ MAP 'k 'v → SET ('k * 'v)
val toSetBy : ∀ 'k 'v. (('k * 'v) → ('k * 'v) → ORDERING) → MAP 'k 'v → SET ('k * 'v)

declare ocaml target_rep function toSetBy = 'Pmap.bindings'
let inline {ocaml} toSet = toSetBy setElemCompare
declare isabelle target_rep function toSet = 'map_to_set'
declare hol target_rep function toSet = 'FMAP_TO_SET'
declare coq target_rep function toSet = 'id'
```

```
assert toSet0 : (toSet (empty : MAP NAT BOOL) = {})
assert toSet1 : (toSet (fromList [((2 : NAT), true); (3, true); (4, false)]) =
              {(2, true), (3, true), (4, false)})
assert toSet2 : (toSet (fromList [((2 : NAT), true); (3, true); (2, false); (4, false)]) =
              {(2, false), (3, true), (4, false)})


val domainBy : ∀ 'k 'v. ('k → 'k → ORDERING) → MAP 'k 'v → SET 'k
val domain : ∀ 'k 'v. MapKeyType 'k, SetType 'k ⇒ MAP 'k 'v → SET 'k
declare ocaml target_rep function domain = 'Pmap.domain'
declare isabelle target_rep function domain = 'Map.dom'
declare hol target_rep function domain = 'FDOM'
declare coq target_rep function domainBy = 'fmap_domain_by'
let inline {coq} domain = domainBy setElemCompare

assert domain0 : (domain (empty : MAP NAT BOOL) = {})
assert domain1 : (domain (fromList [((2 : NAT), true); (3, true); (4, false)]) =
              {2, 3, 4})
assert domain2 : (domain (fromList [((2 : NAT), true); (3, true); (2, false); (4, false)]) =
              {2, 3, 4})


val range : ∀ 'k 'v. MapKeyType 'k, SetType 'v ⇒ MAP 'k 'v → SET 'v
val rangeBy : ∀ 'k 'v. ('v → 'v → ORDERING) → MAP 'k 'v → SET 'v

declare ocaml target_rep function rangeBy = 'Pmap.range'
declare hol target_rep function range = 'FRANGE'
declare isabelle target_rep function range = 'Map.ran'
declare coq target_rep function rangeBy = 'fmap_range_by'
let inline {ocaml, coq} range = rangeBy setElemCompare

assert range0 : (range (empty : MAP NAT BOOL) = {})
assert range1 : (range (fromList [((2 : NAT), true); (3, true); (4, false)]) =
              {true, false})
assert range2 : (range (fromList [((2 : NAT), true); (3, true); (4, true)]) = {true})


(* -----------------------------------------------------------------
(* member                                                  *)
(* -----------------------------------------------------------------

val member : ∀ 'k 'v. MapKeyType 'k, SetType 'k, Eq 'k ⇒ 'k → MAP 'k 'v → BOOL
let inline member k m = k ∈ domain m
declare ocaml target_rep function member = 'Pmap.mem'

val notMember : ∀ 'k 'v. MapKeyType 'k, SetType 'k, Eq 'k ⇒ 'k → MAP 'k 'v → BOOL
let inline notMember k m = ¬ (member k m)

assert member_insert1 : (member 16 (insert (16 : NAT) true empty))
assert member_insert2 : (¬ (member 25 (insert (16 : NAT) true empty)))
assert member_insert3 : (member 16 (insert 36 false (insert (16 : NAT) true empty)))

lemma member_empty : (∀ k. ¬ (member k empty))
lemma member_insert : (∀ k k' v m. member k (insert k' v m) = ((k = k') ∨ member k m))

(* -----------------------------------------------------------------
(* Quantification                                          *)
```

```
(* ------------------------------------------------------------------------

val any : ∀ 'k 'v. MapKeyType 'k, Eq 'v ⇒ ('k → 'v → BOOL) → MAP 'k 'v → BOOL
val all : ∀ 'k 'v. MapKeyType 'k, Eq 'v ⇒ ('k → 'v → BOOL) → MAP 'k 'v → BOOL

let all P m =  (∀ k v. (P k v ∧ (lookup k m = Just v)))
let inline any P m = ¬ (all (fun k v → ¬ (P k v)) m)

declare ocaml target_rep function any = 'Pmap.exist'
declare ocaml target_rep function all = 'Pmap.for_all'
declare coq target_rep function all = 'fmap_all'
declare isabelle target_rep function any = 'map_any'
declare isabelle target_rep function all = 'map_all'
declare hol target_rep function all P = 'FEVERY' (uncurry P)

assert any0 : (any (fun _k v → v) (insert 36 false (insert (16 : NAT) true empty)))
assert any1 : (¬ (any (fun _k v → v) (insert 36 false (insert (16 : NAT) false empty))))
assert any2 : (any (fun _k v → ¬ v) (insert 36 false (insert (16 : NAT) true empty)))
assert any3 : (¬ (any (fun _k v → ¬ v) (insert 36 true (insert (16 : NAT) true empty))))

assert all0 : (all (fun _k v → v) (insert 36 true (insert (16 : NAT) true empty)))
assert all1 : (¬ (all (fun _k v → v) (insert 36 true (insert (16 : NAT) false empty))))
assert all2 : (all (fun _k v → ¬ v) (insert 36 false (insert (16 : NAT) false empty)))
assert all3 : (¬ (all (fun _k v → ¬ v) (insert 36 false (insert (16 : NAT) true empty))))


(* ------------------------------------------------------------------------
(* Set-like operations.                                                  *)
(* ------------------------------------------------------------------------
val deleteBy : ∀ 'k 'v. ('k → 'k → ORDERING) → 'k → MAP 'k 'v → MAP 'k 'v
val delete : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → MAP 'k 'v → MAP 'k 'v
val deleteSwap : ∀ 'k 'v. MapKeyType 'k ⇒ MAP 'k 'v → 'k → MAP 'k 'v

declare coq target_rep function deleteBy = 'fmap_delete_by'
declare ocaml target_rep function delete = 'Pmap.remove'
declare isabelle target_rep function delete = 'map_remove'
declare hol target_rep function deleteSwap = infix '\\'
let inline {hol} delete k m = deleteSwap m k
let inline {coq} delete = deleteBy mapKeyCompare
let inline {coq} deleteSwap m k = delete k m

assert delete_insert1 : (¬ (member (5 : NAT) (delete 5 (insert 5 true empty))))
assert delete_insert2 : (member (7 : NAT) (delete 5 (insert 7 true empty)))
assert delete_delete : (null (delete (5 : NAT) (delete (5 : NAT) (insert 5 true empty))))

val union : ∀ 'k 'v. MapKeyType 'k ⇒ MAP 'k 'v → MAP 'k 'v → MAP 'k 'v
declare coq target_rep function union = ('@' 'List.app' '_')
declare ocaml target_rep function union = 'Pmap.union'
declare isabelle target_rep function union = infix '++'
declare hol target_rep function union = 'FUNION'

val unions : ∀ 'k 'v. MapKeyType 'k ⇒ LIST (MAP 'k 'v) → MAP 'k 'v
let inline unions = foldr (union) empty


(* ------------------------------------------------------------------------
(* Maps (in the functor sense).                                          *)
(* ------------------------------------------------------------------------
```

```
val map : ∀ 'k 'v 'w. MapKeyType 'k ⇒ ('v → 'w) → MAP 'k 'v → MAP 'k 'w

declare hol target_rep function map = infix 'o_f'
declare coq target_rep function map = 'fmap_map'
declare ocaml target_rep function map = 'Pmap.map'
declare isabelle target_rep function map = 'map_image'

assert map_0 : (map (fun b → ¬ b) (insert (2 : NAT) true (insert (3 : NAT) false empty)) =
                insert (2 : NAT) false (insert (3 : NAT) true empty))

val mapi : ∀ 'k 'v 'w. MapKeyType 'k ⇒ ('k → 'v → 'w) → MAP 'k 'v → MAP 'k 'w

(* TODO: add Coq *)
declare ocaml target_rep function mapi = 'Pmap.mapi'
declare isabelle target_rep function mapi = 'map_domain_image'
declare compile_message mapi = "Map.mapi is only defined for the ocaml backend"

(* ------------------------------------------------------------------------
(* Cardinality                                                            *)
(* ------------------------------------------------------------------------
val size : ∀ 'k 'v. MapKeyType 'k, SetType 'k ⇒ MAP 'k 'v → NAT
let inline size m = Set.size (domain m)

declare ocaml target_rep function size = 'Pmap.cardinal'
declare hol target_rep function size = 'FCARD'

assert empty_size : (size (empty : MAP NAT BOOL) = 0)
assert singleton_size : (size (singleton (2 : NAT) (3 : NAT)) = 1)

(* instance of SetType *)
let map_setElemCompare cmp x y =
  cmp (toSet x) (toSet y)

instance ∀ α β. SetType α, SetType β, MapKeyType α ⇒ (SetType (MAP α β))
 let setElemCompare x y = map_setElemCompare setElemCompare x y
end
```

12 Relation

```
(* ********************************************************************************
(* A library for binary relations                                                 *)
(* ********************************************************************************

(* ============================================================================== *)
(* Header                                                                         *)
(* ============================================================================== *)

declare {isabelle, ocaml, hol, coq} rename module = lem_relation

open import Bool Basic_classes Tuple Set Num
open import {hol} set_relationTheory

(* ============================================================================== *)
(* The type of relations                                                          *)
(* ============================================================================== *)

type REL_PRED α β = α → β → BOOL
type REL_SET α β = SET (α * β)

(* Binary relations are usually represented as either  sets of pairs (rel_set) or as curried functions (rel_p

type REL α β = REL_SET α β

val relToSet : ∀ α β. SetType α, SetType β ⇒ REL α β → REL_SET α β
val relFromSet : ∀ α β. SetType α, SetType β ⇒ REL_SET α β → REL α β

let inline relToSet s = s
let inline relFromSet r = r

declare tex target_rep function relFromSet r = r

val relEq : ∀ α β. SetType α, SetType β ⇒ REL α β → REL α β → BOOL
let relEq r1 r2 = (relToSet r1 = relToSet r2)

(*instance forall 'a 'b. SetType 'a, SetType 'b => (Eq (rel 'a 'b)) let (=) = relEqend*)

lemma relToSet_inv : (∀ r. (relToSet r) = r)

val relToPred : ∀ α β. SetType α, SetType β, Eq α, Eq β ⇒ REL α β → REL_PRED α β
val relFromPred : ∀ α β. SetType α, SetType β, Eq α, Eq β ⇒ SET α → SET β → REL_PRED α β →
REL α β

let relToPred r = (fun x y → (x, y) ∈ relToSet r)
let relFromPred xs ys p = Set.filter (fun (x, y) → p x y) (xs × ys)

let inline {hol} relToPred r x y = (x, y) ∈ relToSet r
declare {hol} rename function relToPred = rel_to_pred

assert rel_basic0 : {((2 : NAT), (3 : NAT)), (3, 4)} = relFromPred {2, 3} {1, 2, 3, 4, 5, 6} (fun x y →
y = x + 1)
assert rel_basic1 : relToSet ({((2 : NAT), (3 : NAT)), (3, 4)}) = {(2, 3), (3, 4)}
assert rel_basic2 : relToPred ({((2 : NAT), (3 : NAT)), (3, 4)}) 2 3


(* ============================================================================== *)
```

```
(* Basic Operations                                                    *)
(* ================================================================== *)

(* ------------------------------ *)
(* membership test       *)
(* ------------------------------ *)

val inRel : ∀ α β. SetType α, SetType β, Eq α, Eq β ⇒ α → β → REL α β → BOOL
let inline inRel a b rel = (a, b) ∈ relToSet rel

lemma inRel_set : (∀ s a b. inRel a b (s) = ((a, b) ∈ s))
lemma inRel_pred : (∀ p a b sa sb. inRel a b (relFromPred sa sb p) = p a b ∧ a ∈ sa ∧ b ∈ sb)

assert in_rel0 : (inRel 2 3 ({((2 : NAT), (3 : NAT)), (4, 5)}))
assert in_rel1 : (inRel 4 5 ({((2 : NAT), (3 : NAT)), (4, 5)}))
assert in_rel2 : ¬ (inRel 3 2 ({((2 : NAT), (3 : NAT)), (4, 5)}))
assert in_rel3 : ¬ (inRel 7 4 ({((2 : NAT), (3 : NAT)), (4, 5)}))


(* ------------------------------ *)
(* empty relation        *)
(* ------------------------------ *)

val relEmpty : ∀ α β. SetType α, SetType β ⇒ REL α β
let inline relEmpty = {}
declare tex target_rep function relEmpty = '$\emptyset$'

assert relEmpty0 : relToSet ∅ = ({} : SET (NAT * NAT))
assert relEmpty1 : ¬ (inRel true (2 : NAT) ∅)

(* ------------------------------ *)
(* Insertion           *)
(* ------------------------------ *)

val relAdd : ∀ α β. SetType α, SetType β ⇒ α → β → REL α β → REL α β
let inline relAdd a b r = (insert (a, b) (relToSet r))

assert relAdd0 : inRel (2 : NAT) (3 : NAT) (relAdd 2 3 ∅)
assert relAdd1 : inRel (4 : NAT) (5 : NAT) (relAdd 2 3 (relAdd 4 5 ∅))
assert relAdd2 : ¬ (inRel (2 : NAT) (5 : NAT) (relAdd 2 3 (relAdd 4 5 ∅)))
assert relAdd3 : ¬ (inRel (4 : NAT) (9 : NAT) (relAdd 2 3 (relAdd 4 5 ∅)))

lemma in_relAdd : (∀ a b a' b' r. inRel a b (relAdd a' b' r) =
  ((a = a') ∧ (b = b')) ∨ inRel a b r)


(* ------------------------------ *)
(* Identity relation     *)
(* ------------------------------ *)

val relIdOn : ∀ α. SetType α, Eq α ⇒ SET α → REL α α
let relIdOn s = relFromPred s s (=)

val relId : ∀ α. SetType α, Eq α ⇒ REL α α
let ~{coq, ocaml, isabelle} relId = {(x, x) | ∀ x | true}

declare isabelle target_rep function relId = 'Id'
```

lemma $relId\_spec$: $(\forall\ x\ y\ s.\ (\text{inRel}\ x\ y\ (\text{relIdOn}\ s) \longleftrightarrow (x \in s \wedge (x = y))))$

assert rel_id_0 : $\text{inRel}\ (0 : \text{NAT})\ 0\ (\text{relIdOn}\ \{0, 1, 2, 3\})$
assert rel_id_1 : $\text{inRel}\ (2 : \text{NAT})\ 2\ (\text{relIdOn}\ \{0, 1, 2, 3\})$
assert rel_id_2 : $\neg\ (\text{inRel}\ (5 : \text{NAT})\ 5\ (\text{relIdOn}\ \{0, 1, 2, 3\}))$
assert rel_id_3 : $\neg\ (\text{inRel}\ (0 : \text{NAT})\ 2\ (\text{relIdOn}\ \{0, 1, 2, 3\}))$

```
(* -------------------------- *)
(* relation union          *)
(* -------------------------- *)
```

val $relUnion$: $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta$
let inline $relUnion\ r_1\ r_2\ =\ ((\text{relToSet}\ r_1) \cup (\text{relToSet}\ r_2))$
declare tex target_rep function relUnion = infix '$\cup$'

lemma in_rel_union : $(\forall\ a\ b\ r_1\ r_2.\ \text{inRel}\ a\ b\ (r_1 \cup r_2) = \text{inRel}\ a\ b\ r_1 \vee \text{inRel}\ a\ b\ r_2)$
assert rel_union_0 : (relAdd $(2 : \text{NAT})$ true $\emptyset$) $\cup$ (relAdd 5 false $\emptyset$) =
{(5, false), (2, true)}

```
(* -------------------------- *)
(* relation intersection   *)
(* -------------------------- *)
```

val $relIntersection$: $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta$
let inline $relIntersection\ r_1\ r_2\ =\ ((\text{relToSet}\ r_1) \cap (\text{relToSet}\ r_2))$
declare tex target_rep function relIntersection = infix '$\cap$'

lemma in_rel_inter : $(\forall\ a\ b\ r_1\ r_2.\ \text{inRel}\ a\ b\ (r_1 \cap r_2) = \text{inRel}\ a\ b\ r_1 \wedge \text{inRel}\ a\ b\ r_2)$
assert rel_inter_0 : (relAdd $(2 : \text{NAT})$ true (relAdd 7 false $\emptyset$)) $\cap$
(relAdd 7 false (relAdd 2 false $\emptyset$)) =
{(7, false)}

```
(* -------------------------- *)
(* Relation Composition    *)
(* -------------------------- *)
```

val $relComp$: $\forall\ \alpha\ \beta\ \gamma.\ SetType\ \alpha,\ SetType\ \beta,\ SetType\ \gamma,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \beta\ \gamma\ \rightarrow\ \text{REL}\ \alpha\ \gamma$

let $relComp\ r_1\ r_2 =\ \{(e_1,\ e_3) \mid \forall\ (e_1,\ e_2) \in (\text{relToSet}\ r_1)\ (e_2',\ e_3) \in (\text{relToSet}\ r_2) \mid e_2 = e_2'\}$

declare hol target_rep function relComp = 'rcomp'
declare $isabelle$ target_rep function relComp = infix 'O'

lemma rel_comp_1 : $(\forall\ r_1\ r_2\ e_1\ e_2\ e_3.\ (\text{inRel}\ e_1\ e_2\ r_1 \wedge \text{inRel}\ e_2\ e_3\ r_2) \longrightarrow \text{inRel}\ e_1\ e_3\ (\text{relComp}\ r_1\ r_2))$
lemma $\sim\{coq,\ ocaml\}$ rel_comp_2 : $(\forall\ r.\ (\text{relComp}\ r\ \text{relId} = r) \wedge (\text{relComp relId}\ r = r))$
lemma rel_comp_3 : $(\forall\ r.\ (\text{relComp}\ r\ \emptyset = \emptyset) \wedge (\text{relComp}\ \emptyset\ r = \emptyset))$

assert rel_comp_0 : $(\text{relComp}\ (\{((2 : \text{NAT}),\ (4 : \text{NAT})),\ (2,\ 8)\})\ (\{(4,\ (3 : \text{NAT})),\ (2,\ 8)\}) =$
$\{(2,\ 3)\})$

```
(* -------------------------- *)
(* restrict                *)
(* -------------------------- *)
```

val $relRestrict$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha$
let $relRestrict\ r\ s =\ (\{\ (a,\ b) \mid \forall\ a \in s\ b \in s \mid \text{inRel}\ a\ b\ r\ \})$

declare hol target_rep function relRestrict = 'rrestrict'

```
assert rel_restrict_0 : (relRestrict ({((2 : NAT), (4 : NAT)), (2, 2), (2, 8)}) {2, 8} =
                {(2, 8), (2, 2)})

lemma rel_restrict_empty : (∀ r. relRestrict r {} = ∅)
lemma rel_restrict_rel_empty : (∀ s. relRestrict ∅ s = ∅)
lemma rel_restrict_rel_add : (∀ r x y s. relRestrict (relAdd x y r) s =
  if ((x ∈ s) ∧ (y ∈ s)) then relAdd x y (relRestrict r s) else relRestrict r s)


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* Converse                                               *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val relConverse : ∀ α β. SetType α, SetType β ⇒ REL α β → REL β α
let relConverse r = (Set.map swap (relToSet r))

declare {hol} rename function relConverse = lem_converse
declare isabelle target_rep function relConverse = 'converse'

assert rel_converse_0 : relConverse ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) =
                {(3, 2), (4, 3), (5, 4)}
lemma rel_converse_empty : relConverse ∅ = ∅
lemma rel_converse_add : ∀ x y r. relConverse (relAdd x y r) = relAdd y x (relConverse r)
lemma rel_converse_converse : ∀ r. relConverse (relConverse r) = r


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* domain                                                 *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val relDomain : ∀ α β. SetType α, SetType β ⇒ REL α β → SET α
let relDomain r = Set.map (fun x → fst x) (relToSet r)

declare tex target_rep function relDomain = 'dom'
declare hol target_rep function relDomain = 'domain'
declare isabelle target_rep function relDomain = 'Domain'

assert rel_domain_0 : dom ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) = {2, 3, 4}
assert rel_domain_1 : dom ({((5 : NAT), (3 : NAT)), (3, 4), (4, 5)}) = {3, 4, 5}
assert rel_domain_2 : dom ({((3 : NAT), (3 : NAT)), (3, 4), (4, 5)}) = {3, 4}

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* range                                                  *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val relRange : ∀ α β. SetType α, SetType β ⇒ REL α β → SET β
let relRange r = Set.map (fun x → snd x) (relToSet r)

declare tex target_rep function relRange = 'rng'
declare hol target_rep function relRange = 'range'
declare isabelle target_rep function relRange = 'Range'

assert rel_range_0 : rng ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) = {3, 4, 5}
assert rel_range_1 : rng ({((5 : NAT), (6 : NAT)), (3, 4), (4, 5)}) = {4, 5, 6}
assert rel_range_2 : rng ({((3 : NAT), (5 : NAT)), (3, 4), (4, 5)}) = {4, 5}
```

```
(* ------------------------------ *)
(* field / definedOn      *)
(*                        *)
(* avoid the keyword field *)
(* ------------------------------ *)

val relDefinedOn : ∀ α. SetType α ⇒ REL α α → SET α
let inline relDefinedOn r = ((dom r) ∪ (rng r))

declare {hol} rename function relDefinedOn = rdefined_on

assert rel_field_0 : relDefinedOn ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) = {2, 3, 4, 5}
assert rel_field_1 : relDefinedOn ({((5 : NAT), (6 : NAT)), (3, 4), (4, 5)}) = {3, 4, 5, 6}
assert rel_field_2 : relDefinedOn ({((3 : NAT), (5 : NAT)), (3, 4), (4, 5)}) = {3, 4, 5}

(* ------------------------------ *)
(* relOver              *)
(*                      *)
(* avoid the keyword field *)
(* ------------------------------ *)

val relOver : ∀ α. SetType α ⇒ REL α α → SET α → BOOL
let relOver r s = ((relDefinedOn r) ⊆ s)

declare {hol} rename function relOver = rel_over

assert rel_over_0 : relOver ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) {2, 3, 4, 5}
assert rel_over_1 : ¬ (relOver ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) {3, 4, 5})

lemma rel_over_empty : ∀ s. relOver ∅ s
lemma rel_over_add : ∀ x y s r. relOver (relAdd x y r) s = (x ∈ s ∧ y ∈ s ∧ relOver r s)


(* ------------------------------ *)
(* apply a relation       *)
(* ------------------------------ *)

(* Given a relation r and a set s, relApply r s applies s to r, i.e.  it returns the set of all value reachable

val relApply : ∀ α β. SetType α, SetType β, Eq α ⇒ REL α β → SET α → SET β
let relApply r s = { y | ∀ (x, y) ∈ (relToSet r) | x ∈ s }
declare {hol} rename function relApply = rapply
declare isabelle target_rep function relApply = 'Image'

assert rel_apply_0 : relApply ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) {2, 3} = {3, 4}
assert rel_apply_1 : relApply ({((2 : NAT), (3 : NAT)), (3, 7), (3, 5)}) {2, 3} = {3, 5, 7}

lemma rel_apply_empty_set : ∀ r. relApply r {} = {}
lemma rel_apply_empty : ∀ s. relApply ∅ s = {}
lemma rel_apply_add : ∀ x y s r. relApply (relAdd x y r) s = (if (x ∈ s) then (insert y (relApply r s)) else relApply r s)



(* ========================================================================== *)
(* Properties                                                 *)
(* ========================================================================== *)
```

```
(* -------------------------- *)
(* subrel             *)
(* -------------------------- *)

val isSubrel : ∀ α β. SetType α, SetType β, Eq α, Eq β ⇒ REL α β → REL α β → BOOL
let inline isSubrel r1 r2 = (relToSet r1) ⊆ (relToSet r2)

declare tex target_rep function isSubrel = infix '$\subseteq$'

lemma is_subrel_empty : ∀ r. ∅ ⊆ r
lemma is_subrel_empty2 : ∀ r. r ⊆ ∅ = (r = ∅)
lemma is_subrel_add : ∀ x y r1 r2. (relAdd x y r1) ⊆ r2 = (inRel x y r2 ∧ r1 ⊆ r2)

assert is_subrel0 : ∅ ⊆ ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)})
assert is_subrel1 : ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) ⊆ ({(2, 3), (3, 4), (4, 5)})
assert is_subrel2 : ({((2 : NAT), (3 : NAT)), (4, 5)}) ⊆ ({(2, 3), (3, 4), (4, 5)})
assert is_subrel3 : ¬ (({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)}) ⊆ ({(2, 3), (4, 5)}))

(* -------------------------- *)
(* reflexivity         *)
(* -------------------------- *)

val isReflexiveOn : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → BOOL
let isReflexiveOn r s = (∀ e ∈ s. inRel e e r)

declare {hol} rename function isReflexiveOn = lem_is_reflexive_on

val isReflexive : ∀ α. SetType α, Eq α ⇒ REL α α → BOOL
let ~{ocaml, coq} isReflexive r = (∀ e. inRel e e r)

declare {hol} rename function isReflexive = lem_is_reflexive
declare isabelle target_rep function isReflexive = 'refl'

assert is_reflexive_on0 : isReflexiveOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}) {2, 3}
assert is_reflexive_on1 : ¬ (isReflexiveOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}) {2, 4, 3})
assert is_reflexive_on2 : ¬ (isReflexiveOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}) {5, 2})


(* -------------------------- *)
(* irreflexivity       *)
(* -------------------------- *)

val isIrreflexiveOn : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → BOOL
let isIrreflexiveOn r s = (∀ e ∈ s. ¬ (inRel e e r))

declare hol target_rep function isIrreflexiveOn = 'irreflexive'

val isIrreflexive : ∀ α. SetType α, Eq α ⇒ REL α α → BOOL
let isIrreflexive r = (∀ (e1, e2) ∈ (relToSet r). ¬ (e1 = e2))

declare {hol} rename function isIrreflexive = lem_is_irreflexive
declare isabelle target_rep function isIrreflexive = 'irrefl'

assert is_irreflexive_on0 : isIrreflexiveOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}) {4}
assert is_irreflexive_on1 : ¬ (isIrreflexiveOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}) {2, 4})
assert is_irreflexive_on2 : ¬ (isIrreflexiveOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}) {5, 2})
assert is_irreflexive_on3 : isIrreflexiveOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}) {5, 4}
```

assert $is\_irreflexive_0$: ¬ (isIrreflexive ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}))
assert $is\_irreflexive_1$: isIrreflexive ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5)})

(* – *)
(* symmetry *)
(* – *)

val $isSymmetricOn$: ∀ α. $SetType$ α, Eq α ⇒ REL α α → SET α → BOOL
let $isSymmetricOn$ r s = (∀ e_1 ∈ s e_2 ∈ s. (inRel e_1 e_2 r) ⟶ (inRel e_2 e_1 r))

declare {hol} rename function isSymmetricOn = lem_is_symmetric_on

val $isSymmetric$: ∀ α. $SetType$ α, Eq α ⇒ REL α α → BOOL
let $isSymmetric$ r = (∀ (e_1, e_2) ∈ relToSet r. inRel e_2 e_1 r)

declare {hol} rename function isSymmetric = lem_is_symmetric
declare $isabelle$ target_rep function isSymmetric = 'sym'

assert $is\_symmetric\_on_0$: isSymmetricOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5), (5, 4)}) {4}
assert $is\_symmetric\_on_1$: isSymmetricOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5), (5, 4)}) {3}
assert $is\_symmetric\_on_2$: ¬ (isSymmetricOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5), (5, 4)}) {3, 4})

assert $is\_symmetric_0$: ¬ (isSymmetric ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}))
assert $is\_symmetric_1$: isSymmetric ({((2 : NAT), (3 : NAT)), (3, 2), (4, 5), (5, 4)})

lemma $is\_symmetric\_empty$: ∀ r. isSymmetricOn r {}
lemma $is\_symmetric\_sing$: ∀ r x. isSymmetricOn r {x}

(* – *)
(* antisymmetry *)
(* – *)

val $isAntisymmetricOn$: ∀ α. $SetType$ α, Eq α ⇒ REL α α → SET α → BOOL
let $isAntisymmetricOn$ r s = (∀ e_1 ∈ s e_2 ∈ s. (inRel e_1 e_2 r) ⟶ (inRel e_2 e_1 r) ⟶ (e_1 = e_2))

declare {hol} rename function isAntisymmetricOn = lem_is_antisymmetric_on

val $isAntisymmetric$: ∀ α. $SetType$ α, Eq α ⇒ REL α α → BOOL
let $isAntisymmetric$ r = (∀ (e_1, e_2) ∈ relToSet r. (inRel e_2 e_1 r) ⟶ (e_1 = e_2))

declare hol target_rep function isAntisymmetric = 'antisym'
declare $isabelle$ target_rep function isAntisymmetric = 'antisym'

assert $is\_antisymmetric\_on_0$: isAntisymmetricOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5), (5, 4)}) {3, 4}

assert $is\_antisymmetric\_on_1$: ¬ (isAntisymmetricOn ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5), (5, 4)}) {4, 5})

assert $is\_antisymmetric_0$: isAntisymmetric ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)})
assert $is\_antisymmetric_1$: ¬ (isAntisymmetric ({((2 : NAT), (3 : NAT)), (3, 2), (4, 5), (2, 4)}))

lemma $is\_antisymmetric\_empty$: ∀ r. isAntisymmetricOn r {}
lemma $is\_antisymmetric\_sing$: ∀ r x. isAntisymmetricOn r {x}

```
(* ---------------------------------- *)
(* transitivity                       *)
(* ---------------------------------- *)

val isTransitiveOn : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → BOOL
let isTransitiveOn r s = (∀ e1 ∈ s e2 ∈ s e3 ∈ s. (inRel e1 e2 r) ⟶ (inRel e2 e3 r) ⟶ (inRel e1 e3 r))

declare {hol} rename function isTransitiveOn = lem_transitive_on

val isTransitive : ∀ α. SetType α, Eq α ⇒ REL α α → BOOL
let isTransitive r = (∀ (e1, e2) ∈ relToSet r e3 ∈ relApply r {e2}. inRel e1 e3 r)

declare hol target_rep function isTransitive = 'transitive'
declare isabelle target_rep function isTransitive = 'trans'

assert is_transitive_on0 : isTransitiveOn ({((2 : NAT), (3 : NAT)), (3, 4), (2, 4), (4, 5), (5, 4)}) {2, 3, 4}
assert is_transitive_on1 : ¬ (isTransitiveOn ({((2 : NAT), (3 : NAT)), (3, 4), (2, 4), (4, 5), (5, 4)}) {2, 3, 4, 5})


assert is_transitive0 : ¬ (isTransitive ({((2 : NAT), (2 : NAT)), (3, 3), (3, 4), (4, 5)}))
assert is_transitive1 : isTransitive ({((2 : NAT), (3 : NAT)), (3, 4), (2, 4) })

(* ---------------------------------- *)
(* total                              *)
(* ---------------------------------- *)

val isTotalOn : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → BOOL
let isTotalOn r s = (∀ e1 ∈ s e2 ∈ s. (inRel e1 e2 r) ∨ (inRel e2 e1 r))

declare {hol} rename function isTotalOn = lem_is_total_on


val isTotal : ∀ α. SetType α, Eq α ⇒ REL α α → BOOL
let ~{ocaml, coq} isTotal r = (∀ e1 e2. (inRel e1 e2 r) ∨ (inRel e2 e1 r))
declare {hol} rename function isTotal = lem_is_total
declare isabelle target_rep function isTotal = 'total'

val isTrichotomousOn : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → BOOL
let isTrichotomousOn r s = (∀ e1 ∈ s e2 ∈ s. (inRel e1 e2 r) ∨ (e1 = e2) ∨ (inRel e2 e1 r))

declare {hol} rename function isTrichotomousOn = lem_is_trichotomous_on

val isTrichotomous : ∀ α. SetType α, Eq α ⇒ REL α α → BOOL
let ~{ocaml, coq} isTrichotomous r = (∀ e1 e2. (inRel e1 e2 r) ∨ (e1 = e2) ∨ (inRel e2 e1 r))

declare {hol} rename function isTrichotomous = lem_is_trichotomous


assert is_total_on0 : isTotalOn ({((2 : NAT), (3 : NAT)), (3, 4), (3, 3), (4, 4)}) {3, 4}
assert is_total_on1 : ¬ (isTotalOn ({((2 : NAT), (3 : NAT)), (3, 4), (3, 3), (4, 4)}) {2, 4})

assert is_trichotomous_on0 : isTrichotomousOn ({((2 : NAT), (3 : NAT)), (3, 4)}) {3, 4}
assert is_trichotomous_on1 : ¬ (isTrichotomousOn ({((2 : NAT), (3 : NAT)), (3, 4)}) {2, 3, 4})


(* ---------------------------------- *)
(* is_single_valued                   *)
(* ---------------------------------- *)
```

val *isSingleValued* : $\forall\ \alpha\ \beta.$ *SetType* α, *SetType* β, *Eq* α, *Eq* $\beta\ \Rightarrow$ REL $\alpha\ \beta\ \rightarrow$ BOOL
let *isSingleValued* $r =$ $(\forall\ (e_1,\ e2a) \in \text{relToSet}\ r\ e2b \in \text{relApply}\ r\ \{e_1\}.\ e2a = e2b)$

declare $\{hol\}$ rename function isSingleValued = lem_is_single_valued

assert $is\_single\_valued_0$: isSingleValued $(\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4)\})$
assert $is\_single\_valued_1$: $\neg$ (isSingleValued $(\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (2, 4)\ ,\ (3,\ 4)\})$)

(* - *)
(* equivalence relation *)
(* - *)

val *isEquivalenceOn* : $\forall\ \alpha.$ *SetType* α, *Eq* $\alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ SET $\alpha\ \rightarrow$ BOOL
let *isEquivalenceOn* $r\ s =$ isReflexiveOn $r\ s \wedge$ isSymmetricOn $r\ s \wedge$ isTransitiveOn $r\ s$

declare $\{hol\}$ rename function isEquivalenceOn = lem_is_equivalence_on

val *isEquivalence* : $\forall\ \alpha.$ *SetType* α, *Eq* $\alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ BOOL
let $\sim\{ocaml, coq\}$ *isEquivalence* $r =$ isReflexive $r \wedge$ isSymmetric $r \wedge$ isTransitive r

declare $\{hol\}$ rename function isEquivalence = lem_is_equivalence

assert $is\_equivalence_0$: isEquivalenceOn $(\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (2, 2),\ (3, 3),\ (4, 4)\})$ $\{2, 3, 4\}$

assert $is\_equivalence_1$: $\neg$ (isEquivalenceOn $(\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (2,\ 4),\ (2, 2),\ (3, 3),\ (4, 4)\})$ $\{2, 3, 4\}$)

assert $is\_equivalence_2$: $\neg$ (isEquivalenceOn $(\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (2, 2),\ (3, 3),\ \})$ $\{2, 3, 4\}$)

(* - *)
(* well founded *)
(* - *)

val *isWellFounded* : $\forall\ \alpha.$ *SetType* α, *Eq* $\alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ BOOL
let $\sim\{ocaml, coq\}$ *isWellFounded* $r =$ $(\forall\ P.\ (\forall\ x.\ (\forall\ y.\ \text{inRel}\ y\ x\ r \longrightarrow P\ x) \longrightarrow P\ x) \longrightarrow (\forall\ x.\ P\ x))$

declare *hol* target_rep function isWellFounded r = 'WF' ('reln_to_rel' r)

(* == *)
(* Orders *)
(* == *)

(* - *)
(* pre - or quasiorders *)
(* - *)

val *isPreorderOn* : $\forall\ \alpha.$ *SetType* α, *Eq* $\alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ SET $\alpha\ \rightarrow$ BOOL
let *isPreorderOn* $r\ s =$ isReflexiveOn $r\ s \wedge$ isTransitiveOn $r\ s$

declare $\{hol\}$ rename function isPreorderOn = lem_is_preorder_on

val *isPreorder* : $\forall\ \alpha.$ *SetType* α, *Eq* $\alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ BOOL

```
let ~{ocaml, coq} isPreorder r =  isReflexive r ∧ isTransitive r

declare {hol} rename function isPreorder = lem_is_preorder

assert is_preorder_0 : isPreorderOn ({((2 : NAT), (3 : NAT)), (3, 2), (2, 2), (3, 3), (4, 4)}) {2, 3, 4}
assert is_preorder_1 : ¬ (isPreorderOn ({((2 : NAT), (3 : NAT)), (2, 2), (3, 3)}) {2, 3, 4})
assert is_preorder_2 : ¬ (isPreorderOn ({((2 : NAT), (3 : NAT)), (3, 4), (2, 2), (3, 3), (4, 4)}) {2, 3, 4})


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* partial orders       *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val isPartialOrderOn : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → BOOL
let isPartialOrderOn r s =  isReflexiveOn r s ∧ isTransitiveOn r s ∧ isAntisymmetricOn r s

declare {hol} rename function isPartialOrderOn = lem_is_partial_order_on

assert is_partialorder_0 : isPartialOrderOn ({((2 : NAT), (3 : NAT)), (2, 2), (3, 3), (4, 4)}) {2, 3, 4}
assert is_partialorder_1 : ¬ (isPartialOrderOn ({((2 : NAT), (3 : NAT)), (3, 2), (2, 2), (3, 3), (4, 4)}) {2, 3, 4})

assert is_partialorder_2 : ¬ (isPartialOrderOn ({((2 : NAT), (3 : NAT)), (2, 2), (3, 3)}) {2, 3, 4})
assert is_partialorder_3 : ¬ (isPartialOrderOn ({((2 : NAT), (3 : NAT)), (3, 4), (2, 2), (3, 3), (4, 4)}) {2, 3, 4})



val isStrictPartialOrderOn : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → BOOL
let isStrictPartialOrderOn r s =  isIrreflexiveOn r s ∧ isTransitiveOn r s

declare {hol} rename function isStrictPartialOrderOn = lem_is_strict_partial_order_on

lemma isStrictPartialOrderOn_antisym : (∀ r s. isStrictPartialOrderOn r s ⟶ isAntisymmetricOn r s)

assert is_strict_partialorder_on_0 : isStrictPartialOrderOn ({((2 : NAT), (3 : NAT))}) {2, 3, 4}
assert is_strict_partialorder_on_1 : isStrictPartialOrderOn ({((2 : NAT), (3 : NAT)), (3, 4), (2, 4)}) {2, 3, 4}

assert is_strict_partialorder_on_2 : ¬ (isStrictPartialOrderOn ({((2 : NAT), (3 : NAT)), (3, 4)}) {2, 3, 4})
assert is_strict_partialorder_on_3 : ¬ (isStrictPartialOrderOn ({((2 : NAT), (3 : NAT)), (3, 2)}) {2, 3, 4})
assert is_strict_partialorder_on_4 : ¬ (isStrictPartialOrderOn ({((2 : NAT), (3 : NAT)), (2, 2)}) {2, 3, 4})


val isStrictPartialOrder : ∀ α. SetType α, Eq α ⇒ REL α α → BOOL
let isStrictPartialOrder r =  isIrreflexive r ∧ isTransitive r

declare {hol} rename function isStrictPartialOrder = lem_is_strict_partial_order

assert is_strict_partialorder_0 : isStrictPartialOrder ({((2 : NAT), (3 : NAT))})
assert is_strict_partialorder_1 : isStrictPartialOrder ({((2 : NAT), (3 : NAT)), (3, 4), (2, 4)})
assert is_strict_partialorder_2 : ¬ (isStrictPartialOrder ({((2 : NAT), (3 : NAT)), (3, 4)}))
assert is_strict_partialorder_3 : ¬ (isStrictPartialOrder ({((2 : NAT), (3 : NAT)), (3, 2)}))
assert is_strict_partialorder_4 : ¬ (isStrictPartialOrder ({((2 : NAT), (3 : NAT)), (2, 2)}))

val isPartialOrder : ∀ α. SetType α, Eq α ⇒ REL α α → BOOL
let ~{ocaml, coq} isPartialOrder r =  isReflexive r ∧ isTransitive r ∧ isAntisymmetric r

declare {hol} rename function isPartialOrder = lem_is_partial_order

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
```

(* total / linear orders *)
(* −−−−−−−−−−−−−−−−−−−−−−−−− *)

val *isTotalOrderOn* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ SET $\alpha\ \rightarrow$ BOOL
let *isTotalOrderOn* $r\ s =$ isPartialOrderOn $r\ s \wedge$ isTotalOn $r\ s$

declare {*hol*} rename function isTotalOrderOn = lem_is_total_order_on

val *isStrictTotalOrderOn* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ SET $\alpha\ \rightarrow$ BOOL
let *isStrictTotalOrderOn* $r\ s =$ isStrictPartialOrderOn $r\ s \wedge$ isTrichotomousOn $r\ s$

declare {*hol*} rename function isStrictTotalOrderOn = lem_is_strict_total_order_on

val *isTotalOrder* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ BOOL
let ∼{*ocaml*, *coq*} *isTotalOrder* $r =$ isPartialOrder $r \wedge$ isTotal r

declare {*hol*} rename function isTotalOrder = lem_is_total_order

val *isStrictTotalOrder* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ BOOL
let ∼{*ocaml*, *coq*} *isStrictTotalOrder* $r =$ isStrictPartialOrder $r \wedge$ isTrichotomous r

declare {*hol*} rename function isStrictTotalOrder = lem_is_strict_total_order

assert $is\_totalorder\_on_0$: isTotalOrderOn ({((2 : NAT), (3 : NAT)), (2, 2), (3, 3), (4, 4)}) {2, 3}
assert $is\_totalorder\_on_1$: ¬ (isTotalOrderOn ({((2 : NAT), (3 : NAT)), (2, 2), (3, 3), (4, 4)}) {2, 3, 4})
assert $is\_totalorder\_on_2$: ¬ (isTotalOrderOn ({((2 : NAT), (3 : NAT))}) {2, 3})

assert $is\_strict\_totalorder\_on_0$: isStrictTotalOrderOn ({((2 : NAT), (3 : NAT))}) {2, 3}
assert $is\_strict\_totalorder\_on_1$: ¬ (isStrictTotalOrderOn ({((2 : NAT), (3 : NAT))}) {2, 3, 4})

(* == *)
(* closures *)
(* == *)

(* −−−−−−−−−−−−−−−−−−−−−−−−− *)
(* transitive closure *)
(* −−−−−−−−−−−−−−−−−−−−−−−−− *)

val *transitiveClosure* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ REL $\alpha\ \alpha$
val *transitiveClosureByEq* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow$ BOOL) $\rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ REL $\alpha\ \alpha$
val *transitiveClosureByCmp* : $\forall\ \alpha.\ (\alpha\ *\ \alpha\ \rightarrow\ \alpha\ *\ \alpha\ \rightarrow$ ORDERING) $\rightarrow$ REL $\alpha\ \alpha\ \rightarrow$ REL $\alpha\ \alpha$

declare *ocaml* target_rep function transitiveClosureByCmp = 'Pset.tc'
declare *hol* target_rep function transitiveClosure = 'tc'
declare *isabelle* target_rep function transitiveClosure = 'trancl'
declare *coq* target_rep function transitiveClosureByEq = 'set_tc'

let inline {*coq*} *transitiveClosure* = transitiveClosureByEq (=)
let inline {*ocaml*} *transitiveClosure* = transitiveClosureByCmp setElemCompare

lemma $transitiveClosure\_spec_1$: $(\forall\ r.\ r \subseteq$ (transitiveClosure r))
lemma $transitiveClosure\_spec_2$: $(\forall\ r.$ isTransitive (transitiveClosure r))
lemma $transitiveClosure\_spec_3$: $(\forall\ r_1\ r_2.$ ((isTransitive $r_2) \wedge (r_1 \subseteq r_2)) \longrightarrow$ (transitiveClosure $r_1) \subseteq r_2$)
lemma $transitiveClosure\_spec_4$: $(\forall\ r.$ isTransitive $r \longrightarrow$ (transitiveClosure $r = r$))

```
assert transitive_closure_0 : (transitiveClosure ({((2 : NAT), (3 : NAT)), (3, 4)}) =
                        {(2, 3), (2, 4), (3, 4)})
assert transitive_closure_1 : (transitiveClosure ({((2 : NAT), (3 : NAT)), (3, 4), (4, 5), (7, 9)}) =
                        {(2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5), (7, 9)})


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* transitive closure step *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val transitiveClosureAdd : ∀ α. SetType α, Eq α ⇒ α → α → REL α α → REL α α

let transitiveClosureAdd x y r =
  ((relAdd x y r) ∪ (({(x, z) | ∀ z ∈ rng r | inRel y z r}) ∪
    ({(z, y) | ∀ z ∈ dom r | inRel z x r})))

declare {hol} rename function transitiveClosureAdd = tc_insert

lemma transitive_closure_add_thm : ∀ x y r. isTransitive r ⟶ (transitiveClosureAdd x y r = transitiveClosure (relAdd x y r)


assert transitive_closure_add_0 : transitiveClosureAdd (2 : NAT) (3 : NAT) {} = {(2, 3)}
assert transitive_closure_add_1 : transitiveClosureAdd (3 : NAT) (4 : NAT) {(2, 3)} = {(2, 3), (3, 4), (2, 4)}
assert transitive_closure_add_2 : transitiveClosureAdd (4 : NAT) (5 : NAT) {(2, 3), (3, 4), (2, 4)} =
                          {(2, 3), (3, 4), (2, 4), (4, 5), (2, 5), (3, 5)}


(* ========================================================================== *)
(* reflexive closure                                                          *)
(* ========================================================================== *)

val reflexiveTransitiveClosureOn : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → REL α α
let reflexiveTransitiveClosureOn r s = transitiveClosure (r ∪ (relIdOn s))
declare {hol} rename function reflexiveTransitiveClosureOn = reflexive_transitive_closure_on

assert reflexive_transitive_closure_0 : (reflexiveTransitiveClosureOn ({((2 : NAT), (3 : NAT)), (3, 4)}) {2, 3, 4} =
                        {(2, 3), (2, 4), (3, 4), (2, 2), (3, 3), (4, 4)})


val reflexiveTransitiveClosure : ∀ α. SetType α, Eq α ⇒ REL α α → REL α α
let ~{ocaml, coq} reflexiveTransitiveClosure r = transitiveClosure (r ∪ relId)


(* ========================================================================== *)
(* inverse of closures                                                        *)
(* ========================================================================== *)

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* without transitve edges *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val withoutTransitiveEdges : ∀ α. SetType α, Eq α ⇒ REL α α → REL α α
let withoutTransitiveEdges r =
  let tc = transitiveClosure r in
 {(a, c) | ∀ (a, c) ∈ r
 | ∀ b ∈ rng r. a ≠ b ∧ b ≠ c ⟶ ¬ ((a, b) ∈ tc ∧ (b, c) ∈ tc)}
```

declare *isabelle* target_rep function withoutTransitiveEdges = `'LemExtraDefs.without_trans_edges'`

lemma *trancl_withoutTransitiveEdges_thm* : $\forall\ r.$ finite $r \longrightarrow$
transitiveClosure (withoutTransitiveEdges r) = transitiveClosure r

assert *withoutTransitiveEdges*$_0$: withoutTransitiveEdges $\{((0 : \text{NAT}),\ 1)\}$ = $\{((0 : \text{NAT}),\ 1)\}$
assert *withoutTransitiveEdges*$_1$: withoutTransitiveEdges $\{((0 : \text{NAT}),\ 1),\ (1,\ 2),\ (0,\ 2)\}$ =
$\{((0 : \text{NAT}),\ 1),\ (1,\ 2)\}$
assert *withoutTransitiveEdges*$_2$: withoutTransitiveEdges $\{((0 : \text{NAT}),\ 1),\ (1,\ 2),\ (2,\ 3),\ (0,\ 3)\}$ =
$\{((0 : \text{NAT}),\ 1),\ (1,\ 2),\ (2,\ 3)\}$
assert *withoutTransitiveEdges*$_3$: withoutTransitiveEdges $\{((0 : \text{NAT}),\ 0),\ (0,\ 1)\}$ =
$\{((0 : \text{NAT}),\ 0),\ (0,\ 1)\}$

13 Sorting

```
(*****************************************************************************
(* A library for sorting lists                                               *)
(*                                                                           *)
(* It mainly follows the Haskell List - library                              *)
(*****************************************************************************



(* ========================================================================= *)
(* Header                                                                    *)
(* ========================================================================= *)

declare {isabelle, hol, ocaml, coq} rename module = lem_sorting

open import Bool Basic_classes Maybe List Num

open import {isabelle} HOL - Library.Permutation
open import {coq} Coq.Lists.List
open import {hol} sortingTheory permLib
open import {isabelle} $LIB_DIR/Lem

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* permutations                                              *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val isPermutation : ∀ α. Eq α ⇒ LIST α → LIST α → BOOL
val isPermutationBy : ∀ α. (α → α → BOOL) → LIST α → LIST α → BOOL

let rec isPermutationBy eq l1 l2 = match l1 with
  | [] → null l2
  | (x :: xs) → begin
      match deleteFirst (eq x) l2 with
        | Nothing → false
        | Just ys → isPermutationBy eq xs ys
      end
    end
end
declare termination_argument isPermutationBy = automatic
declare {hol} rename function isPermutationBy = PERM_BY

let inline isPermutation = isPermutationBy (=)

declare isabelle target_rep function isPermutation = infix '<~~>'
declare hol target_rep function isPermutation = 'PERM'

assert perm1 : (isPermutation ([] : LIST NAT) [])
assert perm2 : (¬ (isPermutation [(2 : NAT)] []))
assert perm3 : (isPermutation [(2 : NAT); 1; 3; 5; 4] [1; 2; 3; 4; 5])
assert perm4 : (¬ (isPermutation [(2 : NAT); 3; 3; 5; 4] [1; 2; 3; 4; 5]))
assert perm5 : (¬ (isPermutation [(2 : NAT); 1; 3; 5; 4; 3] [1; 2; 3; 4; 5]))
assert perm6 : (isPermutation [(2 : NAT); 1; 3; 5; 4; 3] [1; 2; 3; 3; 4; 5])

lemma isPermutation1 : (∀ l. isPermutation l l)
lemma isPermutation2 : (∀ l1 l2. isPermutation l1 l2 ⟷ isPermutation l2 l1)
lemma isPermutation3 : (∀ l1 l2 l3. isPermutation l1 l2 ⟶ isPermutation l2 l3 ⟶ isPermutation l1 l3)
lemma isPermutation4 : (∀ l1 l2. isPermutation l1 l2 ⟶ (length l1 = length l2))
```

```
lemma isPermutation_5 : (∀ l_1 l_2. isPermutation l_1 l_2 ⟶ (∀ x. elem x l_1 = elem x l_2))



(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* isSorted                *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* isSortedBy R l  checks, whether the list l is sorted by ordering R.  R should represent an order, i.e. it sh

val isSorted : ∀ α. Ord α ⇒ LIST α → BOOL
val isSortedBy : ∀ α. (α → α → BOOL) → LIST α → BOOL

(* DPM : rejigged the definition with a nested match to get past Coq's termination checker. *)
let rec isSortedBy cmp l =  match l with
  | [] → true
  | x_1 :: xs →
    match xs with
      | [] → true
      | x_2 :: _ → (cmp x_1 x_2 ∧ isSortedBy cmp xs)
    end
end
declare termination_argument isSortedBy = automatic

let inline isSorted = isSortedBy (≤)

declare isabelle target_rep function isSortedBy = 'sorted_by'
declare hol target_rep function isSortedBy = 'SORTED'

assert isSorted_1 : (isSorted ([] : LIST NAT))
assert isSorted_2 : (isSorted [(2 : NAT)])
assert isSorted_3 : (isSorted [(2 : NAT); 4; 5])
assert isSorted_4 : (isSorted [(1 : NAT); 2; 2; 4; 4; 8])
assert isSorted_5 : (¬ (isSorted [(3 : NAT); 2]))
assert isSorted_6 : (¬ (isSorted [(1 : NAT); 2; 3; 2; 3; 4; 5]))



(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* insertion sort          *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val insert : ∀ α. Ord α ⇒ α → LIST α → LIST α
val insertBy : ∀ α. (α → α → BOOL) → α → LIST α → LIST α

val insertSort : ∀ α. Ord α ⇒ LIST α → LIST α
val insertSortBy : ∀ α. (α → α → BOOL) → LIST α → LIST α

let rec insertBy cmp e l =  match l with
  | [] → [e]
  | x :: xs → if cmp x e then x :: (insertBy cmp e xs) else (e :: x :: xs)
end
declare termination_argument insertBy = automatic

let inline insert = insertBy (≤)

let insertSortBy cmp l =  List.foldl (fun l e → insertBy cmp e l) [] l
let inline insertSort = insertSortBy (≤)
```

```
declare isabelle target_rep function insertBy = 'insert_sort_insert_by'
declare isabelle target_rep function insertSortBy = 'insert_sort_by'

declare {hol} rename function insertBy = INSERT_SORT_INSERT
declare {hol} rename function insertSortBy = INSERT_SORT

lemma insertBy_1 : (∀ l e cmp. ((∀ x y z. cmp x y ∧ cmp y z ⟶ cmp x z) ∧ isSortedBy cmp l) ⟶ isSortedBy cmp (insertBy

lemma insertBy_2 : (∀ l e cmp. length (insertBy cmp e l) = length l + 1)
lemma insertBy_3 : (∀ l e_1 e_2 cmp. elem e_1 (insertBy cmp e_2 l) = ((e_1 = e_2) ∨ elem e_1 l))

lemma insertSort_1 : (∀ l cmp. isPermutation (insertSort l) l)
lemma insertSort_2 : (∀ l cmp. isSorted (insertSort l))


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* general sorting       *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val sort : ∀ α. Ord α ⇒ LIST α → LIST α
val sortBy : ∀ α. (α → α → BOOL) → LIST α → LIST α
val sortByOrd : ∀ α. (α → α → ORDERING) → LIST α → LIST α

val predicate_of_ord : ∀ α. (α → α → ORDERING) → α → α → BOOL
let predicate_of_ord f x y =
  match f x y with
  | LT → true
  | EQ → true
  | GT → false
 end

let inline sortBy = insertSortBy
declare isabelle target_rep function sortBy = 'sort_by'
declare hol target_rep function sortBy = 'QSORT'
declare ocaml target_rep function sortByOrd = 'List.sort'

let inline {isabelle, hol} sortByOrd f xs = sortBy (predicate_of_ord f) xs
declare coq target_rep function sortByOrd = 'sort_by_ordering'

let inline ~{ocaml} sort = sortBy (≤)
let inline {ocaml} sort = sortByOrd compare


assert sort_1 : (sort ([] : LIST NAT) = [])
assert sort_2 : (sort ([6; 4; 3; 8; 1; 2] : LIST NAT) = [1; 2; 3; 4; 6; 8])
assert sort_3 : (sort ([5; 4; 5; 2; 4] : LIST NAT) = [2; 4; 4; 5; 5])

lemma sort_4 : (∀ l cmp. isPermutation (sort l) l)
lemma sort_5 : (∀ l cmp. isSorted (sort l))
```

14 Function_extra

declare {*isabelle*, *hol*, *ocaml*, *coq*} rename module = lem_function_extra

open import *Maybe Bool Basic_classes Num Function*

open import {*hol*} *lemTheory*
open import {*isabelle*} *$LIB_DIR/Lem*

(∗ − ∗)
(∗ Tests for function ∗)
(∗ − ∗)

(∗ These tests are not written in function itself, because the nat type is not available there, yet ∗)

assert id_0 : id (2 : NAT) = 2
assert id_1 : id (5 : NAT) = 5
assert id_2 : id (2 : NAT) = 2

assert $const_0$: (const (2 : NAT)) true = 2
assert $const_1$: (const (5 : NAT)) false = 5
assert $const_2$: (const (2 : NAT)) (3 : NAT) = 2

assert $comb_0$: (comb (fun (x : NAT) → 3 ∗ x) succ 2 = 9)
assert $comb_1$: (comb succ (fun (x : NAT) → 3 ∗ x) 2 = 7)

assert $apply_0$: ($) (fun ($x$: NAT) → 3 ∗ x) 2 = 6
assert $apply_1$: (fun (x : NAT) → 3 ∗ x) $ 2 = 6

assert $flip_0$: flip (fun (x : NAT) y → x − y) 3 5 = 2
assert $flip_1$: flip (fun (x : NAT) y → x − y) 5 3 = 0

(∗ − ∗)
(∗ getting a unique value ∗)
(∗ − ∗)

val *THE* : ∀ α. (α → BOOL) → MAYBE α
declare *hol* target_rep function THE = '$THE'
declare *ocaml* target_rep function THE = 'THE'
declare *isabelle* target_rep function THE = 'The_opt'

lemma ∼{*coq*} *THE_spec* : (∀ p x. (THE p = Just x) ⟷ ((p x) ∧ (∀ y. p y ⟶ (x = y))))

15 Assert_extra

```
(* ------------------------------------------------------------------------- *)
(* impure functions for signalling     *)
(* catastrophic failure, or function   *)
(* preconditions.                      *)
(* ------------------------------------------------------------------------- *)

declare {isabelle, ocaml, hol, coq} rename module = lem_assert_extra
open import {ocaml} Xstring
open import {hol} stringTheory lemTheory
open import {coq} Coq.Strings.Ascii Coq.Strings.String
open import {isabelle} $LIB_DIR/Lem

(* ------------------------------------------------------------------------- *)
(* failing with a proper error message *)
(* ------------------------------------------------------------------------- *)

val failwith : ∀ α. STRING → α
declare ocaml target_rep function failwith = 'failwith'
declare hol target_rep function failwith = 'failwith'
declare isabelle target_rep function failwith = 'failwith'
declare coq target_rep function failwith s = 'DAEMON'

(* ------------------------------------------------------------------------- *)
(* failing without an error message    *)
(* ------------------------------------------------------------------------- *)

val fail : ∀ α. α
let fail = failwith "fail"
declare ocaml target_rep function fail = 'assert' 'false'

(* ------------------------------------------------------------------------- *)
(* assertions                          *)
(* ------------------------------------------------------------------------- *)

val ensure : BOOL → STRING → UNIT
let ensure test msg =
  if test then
    ()
  else
    failwith msg
```

16 List_extra

```
(*****************************************************************************
(* A library for lists - the non-pure part                                   *)
(*                                                                           *)
(* It mainly follows the Haskell List-library                                *)
(*****************************************************************************



(* ========================================================================= *)
(* Header                                                                    *)
(* ========================================================================= *)


(* rename module to clash with existing list modules of targets  problem: renaming from inside the module i

declare {isabelle, hol, ocaml, coq} rename module = lem_list_extra

open import Bool Maybe Basic_classes Tuple Num List Assert_extra

(* ------------------------------------------------------------------------- *)
(* head of non-empty list                                                    *)
(* ------------------------------------------------------------------------- *)
val head : ∀ α. LIST α → α
let head l = match l with | x :: xs → x | [] → failwith "List_extra.head of empty list" end

declare compile_message head = "head is only defined on non-empty list and should therefore be avoided. Use maching ins

declare hol target_rep function head = 'HD'
declare ocaml target_rep function head = 'List.hd'
declare isabelle target_rep function head = 'List.hd'

assert head_simple_1 : (head [3; 1] = (3 : NAT))
assert head_simple_2 : (head [5; 4] = (5 : NAT))


(* ------------------------------------------------------------------------- *)
(* tail of non-empty list                                                    *)
(* ------------------------------------------------------------------------- *)
val tail : ∀ α. LIST α → LIST α
let tail l = match l with | x :: xs → xs | [] → failwith "List_extra.tail of empty list" end

declare compile_message tail = "tail is only defined on non-empty list and should therefore be avoided. Use maching instea


declare hol target_rep function tail = 'TL'
declare ocaml target_rep function tail = 'List.tl'
declare isabelle target_rep function tail = 'List.tl'

assert tail_simple_1 : (tail [(3 : NAT); 1] = [1])
assert tail_simple_2 : (tail [(5 : NAT)] = [])
assert tail_simple_3 : (tail [(5 : NAT); 4; 3; 2] = [4; 3; 2])

lemma head_tail_cons : (∀ l. length l > 0 ⟶ (l = (head l)::(tail l)))


(* ------------------------------------------------------------------------- *)
```

(* last *)
(* ------------------------------ *)
val *last* : $\forall\ \alpha.$ LIST $\alpha \rightarrow \alpha$
let rec *last* l = match l with | $[x] \rightarrow x$ | $x_1 :: x_2 :: xs \rightarrow$ last $(x_2 :: xs)$ | [] $\rightarrow$ failwith *"List_extra.last of empty list"* end

declare compile_message last = *"last is only defined on non-empty list and should therefore be avoided. Use maching instead*

declare *hol* target_rep function last = 'LAST'
declare *isabelle* target_rep function last = 'List.last'

assert $last\_simple_1$: (last [(3 : NAT); 1] = 1)
assert $last\_simple_2$: (last [(5 : NAT); 4] = 4)

(* ------------------------------ *)
(* init *)
(* ------------------------------ *)

(* All elements of a non − empty list except the last one. *)
val *init* : $\forall\ \alpha.$ LIST $\alpha \rightarrow$ LIST α
let rec *init* l = match l with | $[x] \rightarrow$ [] | $x_1 :: x_2 :: xs \rightarrow x_1$::(init $(x_2$::$xs)$) | [] $\rightarrow$ failwith *"List_extra.init of empty list"* end

declare compile_message init = *"init is only defined on non-empty list and should therefore be avoided. Use maching instead*

declare *hol* target_rep function init = 'FRONT'
declare *isabelle* target_rep function init = 'List.butlast'

assert $init\_simple_1$: (init [(3 : NAT); 1] = [3])
assert $init\_simple_2$: (init [(5 : NAT)] = [])
assert $init\_simple_3$: (init [(5 : NAT); 4; 3; 2] = [5; 4; 3])

lemma *init_last_append* : ($\forall\ l.$ length $l > 0 \longrightarrow$ (l = (init l) ++ [last l]))
lemma *init_last_dest* : ($\forall\ l.$ length $l > 0 \longrightarrow$ (dest_init l = Just (init l, last l)))

(* ------------------------------ *)
(* foldl1 / foldr1 *)
(* ------------------------------ *)

(* folding functions for non − empty lists, which don't take the base case *)
val $foldl_1$: $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow$ LIST $\alpha \rightarrow \alpha$
let $foldl_1\ f\ x\_xs$ = match x_xs with | $(x :: xs) \rightarrow$ foldl $f\ x\ xs$ | [] $\rightarrow$ failwith *"List_extra.foldl1 of empty list"* end

declare compile_message foldl_1 = *"foldl1 is only defined on non-empty lists. Better use foldl or explicit pattern matching."*

val $foldr_1$: $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow$ LIST $\alpha \rightarrow \alpha$
let $foldr_1\ f\ x\_xs$ = match x_xs with | $(x :: xs) \rightarrow$ foldr $f\ x\ xs$ | [] $\rightarrow$ failwith *"List_extra.foldr1 of empty list"* end

declare compile_message foldr_1 = *"foldr1 is only defined on non-empty lists. Better use foldr or explicit pattern matching."*

(* ------------------------------ *)

```
(* nth element                *)
(* ------------------------------------ *)

(* get the nth element of a list *)
val nth : ∀ α. LIST α → NAT → α
let nth l n = match index l n with Just e → e | Nothing → failwith "List_extra.nth" end
declare compile_message fold1 = "nth is undefined for too large indices, use carefully"

declare hol target_rep function nth l n = 'EL' n l
declare ocaml target_rep function nth = 'List.nth'
declare isabelle target_rep function nth = 'List.nth'
declare coq target_rep function nth l n = 'List.nth' n l

assert nth0 : (nth [0;1;2;3;4;5] 0 = (0 : NAT))
assert nth1 : (nth [0;1;2;3;4;5] 1 = (1 : NAT))
assert nth2 : (nth [0;1;2;3;4;5] 2 = (2 : NAT))
assert nth3 : (nth [0;1;2;3;4;5] 3 = (3 : NAT))
assert nth4 : (nth [0;1;2;3;4;5] 4 = (4 : NAT))
assert nth5 : (nth [0;1;2;3;4;5] 5 = (5 : NAT))

lemma nth_index : (∀ l n e. n < length l ⟶ index l n = Just (nth l n))


(* ------------------------------------ *)
(* Find_non_pure              *)
(* ------------------------------------ *)
val findNonPure : ∀ α. (α → BOOL) → LIST α → α
let findNonPure P l = match (find P l) with
  | Just e → e
  | Nothing → failwith "List_extra.findNonPure"
end

declare compile_message findNonPure = "findNonPure is undefined if no element with the property is in the list. Better us



(* ------------------------------------ *)
(* zip same length            *)
(* ------------------------------------ *)

val zipSameLength : ∀ α β. LIST α → LIST β → LIST (α * β)
let rec zipSameLength l1 l2 = match (l1, l2) with
  | (x :: xs, y :: ys) → (x, y) :: zipSameLength xs ys
  | ([], []) → []
  | _ → failwith "List_extra.zipSameLength of different length lists"

end
declare termination_argument zipSameLength = automatic

declare compile_message zipSameLength = "zipSameLength is undefined if the two lists have different lengths"

declare hol target_rep function zipSameLength l1 l2 = 'ZIP' (l1, l2)
declare ocaml target_rep function zipSameLength = 'List.combine'

assert zipSameLength1 : (zipSameLength [(1 : NAT); 2;3;4;5] [(2 : NAT); 3;4;5;6] = [(1, 2);(2, 3);(3, 4);(4, 5);(5, 6)])


val unfoldr : ∀ α β. (α → MAYBE (β * α)) → α → LIST β
```

```
let rec unfoldr f x =
  match f x with
  | Just (y, x') →
      y :: unfoldr f x'
  | Nothing →
      []
  end
```

17 String

```
(* ************************************************************************** *)
(* A library for strings                                                      *)
(* ************************************************************************** *)

(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)

declare {ocaml, isabelle, hol, coq} rename module = lem_string

open import Bool Basic_classes List
open import {ocaml} Xstring
open import {hol} lemTheory stringTheory
open import {coq} Coq.Strings.Ascii Coq.Strings.String

(* ------------------------- *)
(* basic instantiations      *)
(* ------------------------- *)

(* set up the string and char types correctly for the backends and make  sure that parsing and equality check

declare ocaml target_rep type CHAR = 'char'
declare hol target_rep type CHAR = 'char'
declare isabelle target_rep type CHAR = 'char'
declare coq target_rep type CHAR = 'ascii'

declare ocaml target_rep type STRING = 'string'
declare hol target_rep type STRING = 'string'
declare isabelle target_rep type STRING = 'string'
declare coq target_rep type STRING = 'string'

assert char_simple_0 : ¬ (#'0' = ((#'1') : CHAR))
assert char_simple_1 : ¬ (#'X' = #'Y')
assert char_simple_2 : ¬ (#'\xAF' = #'\x00')
assert char_simple_3 : ¬ (#'' = #'@')
assert char_simple_4 : ¬ (#'\\' = #'\n')
assert char_simple_5 : (#'\x20' = #'')
assert char_simple_6 : ¬ ([#'\x20'; #''; #'\x60'; #'\x27';#'~';#'\\'] = [])

assert string_simple_0 : ¬ ("Hello" = ("Goodby" : STRING))
assert string_simple_1 : ¬ ("Hello\nWorld" = "Goodby\x20!")
assert string_simple_2 : ¬ ("123_\\\t-+!?X_&" = "!'")
assert string_simple_3 : ("Hello World" = ("Hello\x20World" : STRING))

(* ------------------------------------------------------------------------- *)
(* translations between strings and char lists *)
(* ------------------------------------------------------------------------- *)

val toCharList : STRING → LIST CHAR
declare ocaml target_rep function toCharList = 'Xstring.explode'
declare hol target_rep function toCharList = 'EXPLODE'
declare isabelle target_rep function toCharList s = ''s
declare coq target_rep function toCharList = 'string_to_char_list' (* TODO : check *)

assert toCharList_0 : (toCharList "Hello" = [#'H'; #'e'; #'l'; #'l'; #'o'])
assert toCharList_1 : (toCharList "H\nA" = [#'H'; #'\n'; #'A'])
```

```
val toString : list char → string
declare ocaml target_rep function toString = 'Xstring.implode'
declare hol target_rep function toString = 'IMPLODE'
declare isabelle target_rep function toString s = ''s
declare coq target_rep function toString = 'string_from_char_list' (* TODO: check *)

assert toString_0 : (toString [#'H'; #'e'; #'l'; #'l'; #'o'] = "Hello")
assert toString_1 : (toString [#'H'; #'\n'; #'A'] = "H\nA")


(* - - - - - - - - - - - - - - - - - - - - - - - - *)
(* generating strings     *)
(* - - - - - - - - - - - - - - - - - - - - - - - - *)

val makeString : nat → char → string
let makeString len c =  toString (replicate len c)
declare ocaml target_rep function makeString = 'String.make'
declare isabelle target_rep function makeString = 'List.replicate'
declare hol target_rep function makeString = 'REPLICATE'
declare coq target_rep function makeString = 'string_make_string'

assert makeString_0 : (makeString 0 #'a' = "")
assert makeString_1 : (makeString 5 #'a' = "aaaaa")
assert makeString_2 : (makeString 3 #'c' = "ccc")

(* - - - - - - - - - - - - - - - - - - - - - - - - *)
(* length              *)
(* - - - - - - - - - - - - - - - - - - - - - - - - *)

val stringLength : string → nat
declare hol target_rep function stringLength = 'STRLEN'
declare ocaml target_rep function stringLength = 'String.length'
declare isabelle target_rep function stringLength = 'List.length'
declare coq target_rep function stringLength = 'String.length' (* TODO: check *)

assert stringLength_0 : (stringLength "" = 0)
assert stringLength_1 : (stringLength "abc" = 3)
assert stringLength_2 : (stringLength "123456" = 6)

(* - - - - - - - - - - - - - - - - - - - - - - - - *)
(* string concatenation   *)
(* - - - - - - - - - - - - - - - - - - - - - - - - *)

val ^ [stringAppend] : string → string → string
let inline stringAppend x y = (toString ((toCharList x) ++ (toCharList y)))
declare ocaml target_rep function stringAppend = infix '^'
declare hol target_rep function stringAppend = 'STRCAT'
declare isabelle target_rep function stringAppend = infix '@'
declare coq target_rep function stringAppend = 'String.append'

assert stringAppend_0 : (^ "Hello" ^ " " "World!" = "Hello World!")


(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* setting up pattern matching *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
```

```
val string_case : ∀ α. STRING → α → (CHAR → STRING → α) → α

let string_case s c_empty c_cons =
  match (toCharList s) with
  | [] → c_empty
  | c :: cs → c_cons c (toString cs)
 end
declare ocaml target_rep function string_case = 'Xstring.string_case'
declare hol target_rep function string_case = 'list_CASE'
declare isabelle target_rep function string_case s c_e c_c = 'case_list' c_e c_c s

val empty_string : STRING
let inline empty_string = ""

assert empty_string_0 : (empty_string = "")
assert empty_string_1 : ¬ (empty_string = "xxx")

val cons_string : CHAR → STRING → STRING
let inline cons_string c s = toString (c :: toCharList s)

assert string_cons_0 : (cons_string #'a' empty_string = "a")
assert string_cons_1 : (cons_string #'x' "yz" = "xyz")

declare ocaml target_rep function cons_string = 'Xstring.cons_string'
declare hol target_rep function cons_string = 'STRING'
declare isabelle target_rep function cons_string = infix '#'


declare pattern_match exhaustive STRING = [ empty_string; cons_string ] string_case

assert string_patterns_0 : (
 match "" with
  | empty_string → true
  | _ → false
 end
)

assert string_patterns_1 : (
 match "abc" with
  | empty_string → ""
  | cons_string c s → (^ makeString 5 c s)
 end = "aaaaabc"
)

val concat : STRING → LIST STRING → STRING
let rec concat sep ss =
  match ss with
  | [] → ""
  | s :: ss' →
    match ss' with
    | [] → s
    | _ → ^ s ^ sep concat sep ss'
    end
 end

declare ocaml target_rep function concat = 'String.concat'
```

18 Num_extra

```
(* ************************************************************ *)
(*                                                              *)
(* A library of additional functions on numbers                 *)
(*                                                              *)
(* ************************************************************ *)

open import Basic_classes
open import Num
open import String
open import Assert_extra

open import {hol} ASCIInumbersTheory

declare {hol, isabelle, ocaml, coq} rename module = lem_num_extra

val naturalOfString : string → $\mathbb{N}$

declare compile_message naturalOfString = "naturalOfString can fail, potentially with an exception, if the string cannot be

declare ocaml target_rep function naturalOfString = 'Nat_big_num.of_string_nat'
declare hol target_rep function naturalOfString = 'toNum'

val integerOfString : string → $\mathbb{Z}$

declare compile_message integerOfString = "integerOfString can fail, potentially with an exception, if the string cannot be

declare ocaml target_rep function integerOfString = 'Nat_big_num.of_string'

val integerOfChar : char → $\mathbb{Z}$

let integerOfChar = function
  | #'0' → 0
  | #'1' → 1
  | #'2' → 2
  | #'3' → 3
  | #'4' → 4
  | #'5' → 5
  | #'6' → 6
  | #'7' → 7
  | #'8' → 8
  | #'9' → 9
  | _ → failwith "integerOfChar: unexpected character"
end

val integerOfStringHelper : list char → $\mathbb{Z}$

let rec integerOfStringHelper s = match s with
  | d :: ds → integerOfChar d + (10 * integerOfStringHelper ds)
  | [] → 0
end

declare {isabelle} termination_argument integerOfStringHelper = automatic

let ~{ocaml, hol} integerOfString s = match String.toCharList s with
  | #'-' :: ds → integerNegate (integerOfStringHelper (List.reverse ds))
  | ds → integerOfStringHelper (List.reverse ds)
```

```
end

let {hol} integerOfString s =  match s with
  | cons_string #'-' s' → integerNegate (integerFromNatural (naturalOfString s'))
  | _ → integerFromNatural (naturalOfString s)
end

assert {ocaml, hol, isabelle} integerOfString_test_1 : (integerOfString "4096" = 4096)
assert {ocaml, hol, isabelle} integerOfString_test_2 : (integerOfString "-4096" = −4096)

(* Truncation integer division (round toward zero) *)
val integerDiv_t : $\MATHBB{Z}$ → $\MATHBB{Z}$ → $\MATHBB{Z}$
declare ocaml target_rep function integerDiv_t = 'Nat_big_num.integerDiv_t'
declare hol target_rep function integerDiv_t = '$/'

(* Truncation modulo *)
val integerRem_t : $\MATHBB{Z}$ → $\MATHBB{Z}$ → $\MATHBB{Z}$
declare ocaml target_rep function integerRem_t = 'Nat_big_num.integerRem_t'
declare hol target_rep function integerRem_t = '$%'

(* Flooring modulo *)
val integerRem_f : $\MATHBB{Z}$ → $\MATHBB{Z}$ → $\MATHBB{Z}$
declare ocaml target_rep function integerRem_f = 'Nat_big_num.integerRem_f'
declare hol target_rep function integerRem_f = '$%'
```

19 Map_extra

```
(*****************************************************************************
(* A library for finite maps                                                 *)
(*****************************************************************************

(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)

declare {isabelle, hol, ocaml, coq} rename module = lem_map_extra

open import Bool Basic_classes Function Assert_extra Maybe List Num Set Map

(* ---------------------------------------------------------------------------
(* find                                                                       *)
(* ---------------------------------------------------------------------------

val find : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → MAP 'k 'v → 'v
let find k m = match (lookup k m) with Just x → x | Nothing → failwith "Map_extra.find" end

declare ocaml target_rep function find = 'Pmap.find'
declare isabelle target_rep function find = 'map_find'
declare hol target_rep function find k m = 'FAPPLY' m k

declare compile_message find = "find is only defined if the key is found. Use lookup instead and handle the not-found case

assert find_insert_1 : (find 16 (insert (16 : NAT) true empty) = true)
assert find_insert_2 : (find 36 (insert 36 false (insert (16 : NAT) true empty)) = false )



(* ---------------------------------------------------------------------------
(* from sets / domain / range                                                 *)
(* ---------------------------------------------------------------------------


val fromSet : ∀ 'k 'v. MapKeyType 'k ⇒ ('k → 'v) → SET 'k → MAP 'k 'v
let fromSet f s = Set_helpers.fold (fun k m → Map.insert k (f k) m) s Map.empty

declare compile_message fromSet = "fromSet only works for finite sets, use carefully."

declare ocaml target_rep function fromSet = 'Pmap.from_set'
declare hol target_rep function fromSet = 'FUN_FMAP'

(*assert fromSet_0 : (fromSet succ (Set.empty : set nat) = Map.empty)assert fromSet_1 : (fromSet succ {(2 : nat)

(* ---------------------------------------------------------------------------
(* fold                                                                       *)
(* ---------------------------------------------------------------------------

val fold : ∀ 'k 'v 'r. MapKeyType 'k, SetType 'k, SetType 'v ⇒ ('k → 'v → 'r → 'r) → MAP 'k 'v →
'r → 'r
let fold f m v = Set_helpers.fold (fun (k, v) r → f k v r) (Map.toSet m) v

declare ocaml target_rep function fold = 'Pmap.fold'

declare compile_message fold = "Map_extra.fold iterates over the elements of the map in a unspecified order"
```

```
(*assert fold_1 : (fold (fun k v a -> (a + k)) (Map.fromList [((2 : nat), (3 : nat)); (3, 4); (4, 5)]) 0 = 9)assert fold

val toList : ∀ 'k 'v. MapKeyType 'k ⇒ MAP 'k 'v → LIST ('k * 'v)

declare ocaml target_rep function toList = 'Pmap.bindings_list'
declare coq target_rep function toList = 'fmap_elements' (* TODO *)
declare hol target_rep function toList = 'MAP_TO_LIST'
declare isabelle target_rep function toList m = 'list_of_set' ('LemExtraDefs.map_to_set' m)
(* declare compile_message toList = "Map_extra.toList is only defined for the ocaml, isabelle and coq backend

(* more 'map' functions *)

(* TODO : this function is in map_extra rather than map just for implementation reasons *)
val mapMaybe : ∀ α β γ. MapKeyType α ⇒ (α → β → MAYBE γ) → MAP α β → MAP α γ
(* OLD : TODO : mapMaybe depends on toList that is not defined for hol and isabelle *)
let mapMaybe f m =
  List.foldl
   (fun m' (k, v) →
     match f k v with
       | Nothing → m'
       | Just v' → Map.insert k v' m'
     end)
   Map.empty
   (toList m)

declare {ocaml, hol, isabelle} rename function mapMaybe = option_map
declare compile_message toList = "Map_extra.mapMaybe is only defined for the ocaml and coq backend"
```

20 Set_extra

```
(* ********************************************************************************
(* A library for sets                                                             *)
(*                                                                                *)
(* It mainly follows the Haskell Set - library                                    *)
(* ********************************************************************************

(* ============================================================================== *)
(* Header                                                                         *)
(* ============================================================================== *)

open import Bool Basic_classes Maybe Function Num List Sorting Set

declare {hol, isabelle, ocaml, coq} rename module = lem_set_extra


(* ------------------------------------------------------------------------------*)
(* set choose (be careful !)  *)
(* ------------------------------------------------------------------------------ *)

val choose : ∀ α. SetType α ⇒ SET α → α
declare compile_message choose = "choose is non-deterministic and only defined for non-empty sets. Its result may differ b


declare hol target_rep function choose = 'CHOICE'
declare isabelle target_rep function choose = 'set_choose'
declare ocaml target_rep function choose = 'Pset.choose'

lemma ~{coq} choose_sing : (∀ x. choose {x} = x)
lemma ~{coq} choose_in : (∀ s. ¬ (null s) ⟶ ((choose s) ∈ s))

assert ~{coq} choose_0 : choose {(2 : NAT)} = 2
assert ~{coq} choose_1 : choose {(5 : NAT)} = 5
assert ~{coq} choose_2 : choose {(6 : NAT)} = 6
assert ~{coq} choose_3 : choose {(6 : NAT), 1, 2} ∈ {6, 1, 2}

(* ------------------------------------------------------------------------------ *)
(* chooseAndSplit        *)
(* ------------------------------------------------------------------------------ *)
(* The idea here is to provide a simple primitive that Lem code can use * to perform its own custom searches
val chooseAndSplit : ∀ α. SetType α, Ord α ⇒ SET α → MAYBE (SET α * α * SET α)
let ~{coq} chooseAndSplit s =
  if s = ∅ then
    Nothing
  else
    let element = choose s in
    let (lt, gt) = Set.split element s in
      Just (lt, element, gt)

declare ocaml target_rep function chooseAndSplit = 'Pset.choose_and_split'
declare coq target_rep function chooseAndSplit = 'choose_and_split'

(* ------------------------------------------------------------------------------*)
(* universal set           *)
(* ------------------------------------------------------------------------------ *)

val universal : ∀ α. SetType α ⇒ SET α
```

```
declare compile_message universal = "universal sets are usually infinite and only available in HOL and Isabelle"


declare hol target_rep function universal = 'UNIV'
declare isabelle target_rep function universal = 'UNIV'

assert {hol} in_univ0 : true ∈ universal
assert {hol} in_univ1 : (1 : NAT) ∈ universal
lemma {hol} in_univ_thm : ∀ x. x ∈ universal


(* ---------------------------------------------------------*)
(* toList                 *)
(* ---------------------------------------------------------- *)

val toList : ∀ α. SetType α ⇒ SET α → LIST α
declare compile_message toList = "toList is only defined on finite sets and the order of the resulting list is unspecified and


declare ocaml target_rep function toList = 'Pset.elements'
declare isabelle target_rep function toList = 'list_of_set'
declare hol target_rep function toList = 'SET_TO_LIST'
declare coq target_rep function toList = 'set_to_list'


assert toList0 : toList ({} : SET NAT) = []
assert toList1 : toList {(6 : NAT), 1, 2} ∈ {[1;2;6], [1;6;2], [2;1;6], [2;6;1], [6;1;2], [6;2;1]}
assert toList2 : toList ({(2 : NAT)} : SET NAT) = [2]


(* ---------------------------------------------------------*)
(* toOrderedList           *)
(* ---------------------------------------------------------- *)

(* "toOrderedList" returns a sorted list. Therefore the result is (given a suitable order) deterministic.  Th

val toOrderedList : ∀ α. (α → α → BOOL) → SET α → LIST α
declare isabelle target_rep function toOrderedListBy = 'ordered_list_of_set'
declare hol target_rep function toOrderedListBy = 'ARB'

val toOrderedList : ∀ α. SetType α, Ord α ⇒ SET α → LIST α
let inline ~{isabelle, ocaml} toOrderedList l = sort (toList l)
let inline {isabelle} toOrderedList = toOrderedListBy (≤)
declare ocaml target_rep function toOrderedList = 'Pset.elements'

declare compile_message toOrderedList = "toOrderedList is only defined on finite sets."

assert toOrderedList0 : toOrderedList ({} : SET NAT) = []
assert toOrderedList1 : toOrderedList {(6 : NAT), 1, 2} = [1;2;6]
assert toOrderedList2 : toOrderedList ({(2 : NAT)} : SET NAT) = [2]

(* ---------------------------------------------------------- *)
(* compare           *)
(* ---------------------------------------------------------- *)

val setCompareBy : ∀ α. (α → α → ORDERING) → SET α → SET α → ORDERING
let {isabelle, hol} setCompareBy cmp ss ts =
  let ss' = toOrderedListBy (fun x y → cmp x y = LT) ss in
```

```
  let ts' = toOrderedListBy (fun x y → cmp x y = LT) ts in
    lexicographicCompareBy cmp ss' ts'

declare coq target_rep function setCompareBy = 'set_compare_by'
declare ocaml target_rep function setCompareBy = 'Pset.compare_by'

val setCompare : ∀ α. SetType α, Ord α ⇒ SET α → SET α → ORDERING
let setCompare = setCompareBy compare

instance ∀ α. SetType α ⇒ (SetType (SET α))
  let setElemCompare = setCompareBy setElemCompare
end

(* ----------------------------------------*)
(* unbounded fixed point      *)
(* ---------------------------------------- *)

(* Is NOT supported by the coq backend! *)
val leastFixedPointUnbounded : ∀ α. SetType α ⇒ (SET α → SET α) → SET α → SET α
let rec leastFixedPointUnbounded f x =
    let fx = f x in
   if fx ⊆ x then x
   else leastFixedPointUnbounded f (fx ∪ x)

declare isabelle target_rep function leastFixedPointUnbounded f s = 'LemExtraDefs.unbounded_lfp' s f

declare compile_message toOrderedList = "leastFixedPointUnbounded is deprecated as it is not supported by all backends (

assert lfp_empty : leastFixedPointUnbounded (map (fun x → x)) ({} : SET NAT) = {}
assert lfp_saturate_neg : leastFixedPointUnbounded (map (fun x → −x)) ({1, 2, 3} : SET INT) = {−3, −2, −1, 1, 2, 3}

assert lfp_saturate_mod : leastFixedPointUnbounded (map (fun x → (2∗x) mod 5)) ({1} : SET NAT) = {1, 2, 3, 4}
```

21 Maybe_extra

```
(*****************************************************************************
(* extra functions for maybe / option                                        *)
(*                                                                           *)
(*****************************************************************************

declare {isabelle, hol, ocaml, coq} rename module = lem_maybe_extra

open import Basic_classes Maybe Assert_extra

(* ------------------------- *)
(* fromJust                  *)
(* ------------------------- *)

val fromJust : ∀ α. MAYBE α → α
let fromJust op = match op with | Just v → v | Nothing → failwith "fromJust of Nothing" end
declare termination_argument fromJust = automatic
declare compile_message fromJust = "fromJust is only defined on Just. Better use 'fromMaybe' or use explicit maching to


declare hol target_rep function fromJust = 'THE'
declare isabelle target_rep function fromJust = 'Option.the'
```

22 String_extra

```
(*************************************************************************
(* String functions                                                      *)
(*************************************************************************

open import Basic_classes
open import Num
open import List
open import String
open import List_extra
open import {hol} stringLib
open import {hol} ASCIInumbersTheory

declare {isabelle, ocaml, hol, coq} rename module = lem_string_extra


(*************************************************************************
(* Character's to numbers                                                *)
(*************************************************************************

val ord : CHAR → NAT
declare hol target_rep function ord = 'ORD'
declare ocaml target_rep function ord = 'Char.code'
(* TODO: The Isabelle and Coq representations are taken from a quick Google  search, they might not be the b
declare isabelle target_rep function ord = 'of_char'
declare coq target_rep function ord = 'nat_of_ascii'

val chr : NAT → CHAR
declare hol target_rep function chr = 'CHR'
declare ocaml target_rep function chr = 'Char.chr'
(* TODO: The Isabelle and Coq representations are taken from a quick Google  search, they might not be the b
declare isabelle target_rep function chr = '(%n.  char_of (n::nat))'
declare coq target_rep function chr = 'ascii_of_nat'

(*************************************************************************
(* Converting to strings                                                 *)
(*************************************************************************

val stringFromNatHelper : NAT → LIST CHAR → LIST CHAR
let rec stringFromNatHelper n acc =
  if n = 0 then
   acc
 else
   stringFromNatHelper (n / 10) (chr (n mod 10 + 48) :: acc)

declare {isabelle} termination_argument stringFromNatHelper = automatic

val stringFromNat : NAT → STRING
let ~{ocaml, hol} stringFromNat n =
  if n = 0 then "0" else toString (stringFromNatHelper n [])

declare ocaml target_rep function stringFromNat = 'string_of_int'
declare hol target_rep function stringFromNat = 'num_to_dec_string'

assert stringFromNat_0 : stringFromNat 0 = "0"
assert stringFromNat_1 : stringFromNat 1 = "1"
assert stringFromNat_2 : stringFromNat 42 = "42"
```

```
val stringFromNaturalHelper : $\mathbb{N}$ → list char → list char
let rec stringFromNaturalHelper n acc =
  if n = 0 then
   acc
 else
   stringFromNaturalHelper (n / 10) (chr (natFromNatural (n mod 10 + 48)) :: acc)

declare {isabelle} termination_argument stringFromNaturalHelper = automatic

val stringFromNatural : $\mathbb{N}$ → string
let ~{ocaml, hol} stringFromNatural n =
  if n = 0 then "0" else toString (stringFromNaturalHelper n [])

declare hol target_rep function stringFromNatural = 'num_to_dec_string'
declare ocaml target_rep function stringFromNatural = 'Nat_big_num.to_string'

assert stringFromNatural_0 : stringFromNatural 0 = "0"
assert stringFromNatural_1 : stringFromNatural 1 = "1"
assert stringFromNatural_2 : stringFromNatural 42 = "42"

val stringFromInt : int → string
let ~{ocaml} stringFromInt i =
  if i < 0 then
   ^ "-" stringFromNat (natFromInt i)
 else
   stringFromNat (natFromInt i)

declare ocaml target_rep function stringFromInt = 'string_of_int'

assert stringFromInt_0 : stringFromInt 0 = "0"
assert stringFromInt_1 : stringFromInt 1 = "1"
assert stringFromInt_2 : stringFromInt 42 = "42"
assert stringFromInt_3 : stringFromInt (−1) = "-1"

val stringFromInteger : $\mathbb{Z}$ → string
let ~{ocaml} stringFromInteger i =
  if i < 0 then
   ^ "-" stringFromNatural (naturalFromInteger i)
 else
   stringFromNatural (naturalFromInteger i)

declare ocaml target_rep function stringFromInteger = 'Nat_big_num.to_string'

assert stringFromInteger_0 : stringFromInteger 0 = "0"
assert stringFromInteger_1 : stringFromInteger 1 = "1"
assert stringFromInteger_2 : stringFromInteger 42 = "42"
assert stringFromInteger_3 : stringFromInteger (−1) = "-1"


(*****************************************************************************
(* List-like operations                                                      *)
(*****************************************************************************

val nth : string → nat → char
let nth s n = List_extra.nth (toCharList s) n

declare hol target_rep function nth l n = 'SUB' (l, n)
```

```
declare ocaml target_rep function nth = 'String.get'

val stringConcat : list string → string
let stringConcat s =
  List.foldr ^ "" s

declare hol target_rep function stringConcat = 'CONCAT'
declare ocaml target_rep function stringConcat s = 'String.concat' "" s

(*****************************************************************************
(* String comparison                                                         *)
(*****************************************************************************

val stringCompare : string → string → ordering

(* TODO: *)
let inline stringCompare x y = EQ (* XXX: broken *)
let inline {ocaml} stringCompare = defaultCompare

declare compile_message stringCompare = "It is highly unclear, what string comparison should do. Do we have abc < ABC


let stringLess x y =  orderingIsLess (stringCompare x y)
let stringLessEq x y =  orderingIsLessEqual (stringCompare x y)
let stringGreater x y =  stringLess y x
let stringGreaterEq x y =  stringLessEq y x

instance (Ord string)
  let compare =  stringCompare
  let < =  stringLess
  let <= =  stringLessEq
  let > =  stringGreater
  let >= =  stringGreaterEq
end

assert {ocaml} string_compare_1 : "abc" < "bbc"
assert {ocaml} string_compare_2 : "abc" ≤ "abc"
assert {ocaml} string_compare_3 : "abc" > "ab"
```

23 Word

```
(* ************************************************************************** *)
(* A generic library for machine words.                                       *)
(* ************************************************************************** *)

declare {isabelle, coq, hol, ocaml} rename module = Lem_word

open import Bool Maybe Num Basic_classes List

open import {isabelle} HOL - Word.Word
open import {hol} wordsTheory wordsLib


(* ========================================================================== *)
(* Define general purpose word, i.e. sequences of bits of arbitrary length    *)
(* ========================================================================== *)

type bitSequence = BitSeq of
  maybe nat * (* length of the sequence, Nothing means infinite length *)
  bool * (* sign of the word, used to fill up after concrete value is exhausted *)
  list bool (* the initial part of the sequence, least significant bit first *)

val bitSeqEq : bitSequence -> bitSequence -> bool
let inline bitSeqEq = unsafe_structural_equality
instance (Eq bitSequence)
  let = = bitSeqEq
  let <> n1 n2 = ¬ (bitSeqEq n1 n2)
end

val boolListFrombitSeq : nat -> bitSequence -> list bool

let rec boolListFrombitSeqAux n s bl =
  if n = 0 then [] else
  match bl with
    | [] -> replicate n s
    | b :: bl' -> b :: (boolListFrombitSeqAux (n-1) s bl')
  end
declare termination_argument boolListFrombitSeqAux = automatic

let boolListFrombitSeq n (BitSeq _ s bl) = boolListFrombitSeqAux n s bl

assert boolListFrombitSeq0 : boolListFrombitSeq 5 (BitSeq Nothing false [true; false; true]) = [true; false; true; false; false]

assert boolListFrombitSeq1 : boolListFrombitSeq 5 (BitSeq Nothing true [true; false; true]) = [true; false; true; true; true]

assert boolListFrombitSeq2 : boolListFrombitSeq 2 (BitSeq Nothing true [true; false; true]) = [true; false]

lemma boolListFrombitSeq_len : ∀ n bs. (List.length (boolListFrombitSeq n bs) = n)


val bitSeqFromBoolList : list bool -> maybe bitSequence
let bitSeqFromBoolList bl =
  match dest_init bl with
    | Nothing -> Nothing
    | Just (bl', s) -> Just (BitSeq (Just (List.length bl)) s bl')
  end
```

```
assert bitSeqFromBoolList_0 : bitSeqFromBoolList [] = Nothing
assert bitSeqFromBoolList_1 : bitSeqFromBoolList [true; false; false] = Just (BitSeq (Just 3) false [true; false])
assert bitSeqFromBoolList_2 : bitSeqFromBoolList [true; false; true] = Just (BitSeq (Just 3) true [true; false])

lemma bitSeqFromBoolList_nothing : ∀ bl. (isNothing (bitSeqFromBoolList bl) ⟷ List.null bl)


(* cleans up the representation of a bitSequence without changing its semantics *)
val cleanBitSeq : BITSEQUENCE → BITSEQUENCE
let cleanBitSeq (BitSeq len s bl) = match len with
  | Nothing → (BitSeq len s (List.reverse (dropWhile ((=) s) (List.reverse bl))))
  | Just n → (BitSeq len s (List.reverse (dropWhile ((=) s) (List.reverse (List.take (n−1) bl)))))
end

assert cleanBitSeq_0 : cleanBitSeq (BitSeq Nothing false [true; false; true; false; false]) = (BitSeq Nothing false [true; false; true])

assert cleanBitSeq_1 : cleanBitSeq (BitSeq Nothing true [true; false; true; false; false]) = (BitSeq Nothing true [true; false; true; fals

assert cleanBitSeq_2 : cleanBitSeq (BitSeq (Just 4) true [true; false; true; false; false]) = (BitSeq (Just 4) true [true; false])



val bitSeqTestBit : BITSEQUENCE → NAT → MAYBE BOOL
let bitSeqTestBit (BitSeq len s bl) pos =
   match len with
    | Nothing → if pos < length bl then index bl pos else Just s
    | Just l → if (pos ≥ l) then Nothing else
               if (pos = (l − 1) ∨ pos ≥ length bl) then Just s else
               index bl pos
 end

val bitSeqSetBit : BITSEQUENCE → NAT → BOOL → BITSEQUENCE
let bitSeqSetBit (BitSeq len s bl) pos v =
   let bl′ = if (pos < length bl) then bl else bl ++ replicate pos s in
  let bl″ = List.update bl′ pos v in
  let bs′ = BitSeq len s bl″ in
  cleanBitSeq bs′


val resizeBitSeq : MAYBE NAT → BITSEQUENCE → BITSEQUENCE
let resizeBitSeq new_len bs =
   let (BitSeq len s bl) = cleanBitSeq bs in
  let shorten_opt = match (new_len, len) with
    | (Nothing, _) → Nothing
    | (Just l_1, Nothing) → Just l_1
    | (Just l_1, Just l_2) → if (l_1 < l_2) then Just l_1 else Nothing
  end in
  match shorten_opt with
    | Nothing → BitSeq new_len s bl
    | Just l_1 → (
       let bl′ = List.take l_1 (bl ++ [s]) in
       match dest_init bl′ with
         | Nothing → (BitSeq len s bl) (* do nothing if size 0 is requested *)
         | Just (bl″, s′) → cleanBitSeq (BitSeq new_len s′ bl″)
 end)
  end

assert resizeBitSeq_0 : (resizeBitSeq Nothing (BitSeq (Just 5) true [false; false]) = (BitSeq Nothing true [false; false]))
```

```
assert resizeBitSeq_1 : (resizeBitSeq (Just 3) (BitSeq Nothing true [false; true; false; false]) = (BitSeq (Just 3) false [false; true]))

assert resizeBitSeq_2 : (resizeBitSeq (Just 3) (BitSeq Nothing false [false; true; true; false]) = (BitSeq (Just 3) true [false]))

assert resizeBitSeq_3 : (resizeBitSeq (Just 3) (BitSeq (Just 10) false [false; true; true; false]) = (BitSeq (Just 3) true [false]))

assert resizeBitSeq_4 : (resizeBitSeq (Just 10) (BitSeq (Just 3) false [false; true; true; false]) = (BitSeq (Just 10) false [false; true]))


val bitSeqNot : bitSequence → bitSequence
let bitSeqNot (BitSeq len s bl) =  BitSeq len (¬ s) (List.map (fun b → ¬ b) bl)

assert bitSeqNot_0 : (bitSeqNot (BitSeq (Just 2) true [false; true])) = BitSeq (Just 2) false [true; false]

val bitSeqBinop : (bool → bool → bool) → bitSequence → bitSequence → bitSequence

val bitSeqBinopAux : (bool → bool → bool) → bool → list bool → bool → list bool →
list bool
let rec bitSeqBinopAux binop s_1 bl_1 s_2 bl_2 =
  match (bl_1, bl_2) with
   | ([], []) → []
   | (b_1 :: bl'_1, []) → (binop b_1 s_2) :: bitSeqBinopAux binop s_1 bl'_1 s_2 []
   | ([], b_2 :: bl'_2) → (binop s_1 b_2) :: bitSeqBinopAux binop s_1 [] s_2 bl'_2
   | (b_1 :: bl'_1, b_2 :: bl'_2) → (binop b_1 b_2) :: bitSeqBinopAux binop s_1 bl'_1 s_2 bl'_2
  end
declare termination_argument bitSeqBinopAux = automatic
declare coq target_rep function bitSeqBinopAux = 'bitSeqBinopAux'

let bitSeqBinop binop bs_1 bs_2 =  (
  let (BitSeq len_1 s_1 bl_1) = cleanBitSeq bs_1 in
  let (BitSeq len_2 s_2 bl_2) = cleanBitSeq bs_2 in

  let len = match (len_1, len_2) with
    | (Just l_1, Just l_2) → Just (max l_1 l_2)
    | _ → Nothing
  end in
  let s = binop s_1 s_2 in
  let bl = bitSeqBinopAux binop s_1 bl_1 s_2 bl_2 in
  cleanBitSeq (BitSeq len s bl)
)

let bitSeqAnd =  bitSeqBinop (∧)
let bitSeqOr =  bitSeqBinop (∨)
let bitSeqXor =  bitSeqBinop xor

val bitSeqShiftLeft : bitSequence → nat → bitSequence
let bitSeqShiftLeft (BitSeq len s bl) n =  cleanBitSeq (BitSeq len s (replicate n false ++ bl))

val bitSeqArithmeticShiftRight : bitSequence → nat → bitSequence
let bitSeqArithmeticShiftRight bs n =
   let (BitSeq len s bl) = cleanBitSeq bs in
  cleanBitSeq (BitSeq len s (drop n bl))

val bitSeqLogicalShiftRight : bitSequence → nat → bitSequence
let bitSeqLogicalShiftRight bs n =
   if (n = 0) then cleanBitSeq bs else
  let (BitSeq len s bl) = cleanBitSeq bs in
```

```
  match len with
    | Nothing → cleanBitSeq (BitSeq len s (drop n bl))
    | Just l → cleanBitSeq (BitSeq len false ((drop n bl) ++ replicate l s))
  end


(* integerFromBoolList sign bl creates an integer from a list of bits  (least significant bit first) and an e
val integerFromBoolList : (BOOL * LIST BOOL) → $\MATHBB{Z}$

let rec integerFromBoolListAux (acc : $\MATHBB{Z}$) (bl : LIST BOOL) =
   match bl with
    | [] → acc
    | (true :: bl') → integerFromBoolListAux ((acc * 2) + 1) bl'
    | (false :: bl') → integerFromBoolListAux (acc * 2) bl'
  end
declare termination_argument integerFromBoolListAux = automatic

let integerFromBoolList (sign, bl) =
    if sign then
     −(integerFromBoolListAux 0 (List.reverseMap (fun b → ¬ b) bl) + 1)
   else integerFromBoolListAux 0 (List.reverse bl)

assert integerFromBoolList_0 : integerFromBoolList (false, [false; true; false]) = 2
assert integerFromBoolList_1 : integerFromBoolList (false, [false; true; false; true]) = 10
assert integerFromBoolList_2 : integerFromBoolList (true, [false; true; false; true]) = −6
assert integerFromBoolList_3 : integerFromBoolList (true, [false; true]) = −2
assert integerFromBoolList_4 : integerFromBoolList (true, [true; false]) = −3

(* [boolListFromInteger i] creates a sign bit and a list of booleans from an integer. The len_opt tells it wher
val boolListFromInteger : $\MATHBB{Z}$ → BOOL * LIST BOOL

let rec boolListFromNatural acc (remainder : $\MATHBB{N}$) =
  if (remainder > 0) then
   (boolListFromNatural (((remainder mod 2) = 1) :: acc)
     (remainder / 2))
 else
   List.reverse acc
declare termination_argument boolListFromNatural = automatic
declare coq target_rep function boolListFromNatural = 'boolListFromNatural'

let boolListFromInteger (i : $\MATHBB{Z}$) =
   if (i < 0) then
    (true, List.map (fun b → ¬ b) (boolListFromNatural [] (naturalFromInteger (−(i + 1)))))
  else
    (false, boolListFromNatural [] (naturalFromInteger i))

assert boolListFromInteger_0 : boolListFromInteger 2 = (false, [false; true])
assert boolListFromInteger_1 : boolListFromInteger 10 = (false, [false; true; false; true])
assert boolListFromInteger_2 : boolListFromInteger (−6) = (true, [false; true; false])
assert boolListFromInteger_3 : boolListFromInteger (−2) = (true, [false])
assert boolListFromInteger_4 : boolListFromInteger (−3) = (true, [true; false])

lemma boolListFromInteger_inverse_1 : (∀ i. integerFromBoolList (boolListFromInteger i) = i)
lemma boolListFromInteger_inverse_2 : (∀ s bl i. boolListFromInteger (integerFromBoolList (s, bl)) =
   (s, List.reverse (dropWhile ((=) s) (List.reverse bl))))


(* [bitSeqFromInteger len_opt i] encodes [i] as a bitsequence with [len_opt] bits. If there are not enough  bits
```

```
val bitSeqFromInteger : maybe nat → $\mathbb{Z}$ → bitSequence
let bitSeqFromInteger len_opt i =
  let (s, bl) = boolListFromInteger i in
 resizeBitSeq len_opt (BitSeq Nothing s bl)

assert bitSeqFromInteger_0 : (bitSeqFromInteger Nothing 5 = BitSeq Nothing false [true; false; true])
assert bitSeqFromInteger_1 : (bitSeqFromInteger (Just 2) 5 = BitSeq (Just 2) false [true])
assert bitSeqFromInteger_2 : (bitSeqFromInteger Nothing (−5) = BitSeq Nothing true [true; true; false])
assert bitSeqFromInteger_3 : (bitSeqFromInteger (Just 3) (−5) = BitSeq (Just 3) false [true; true])
assert bitSeqFromInteger_4 : (bitSeqFromInteger (Just 2) (−5) = BitSeq (Just 2) true [])
assert bitSeqFromInteger_5 : (bitSeqFromInteger (Just 5) (−5) = BitSeq (Just 5) true [true; true; false])


val integerFromBitSeq : bitSequence → $\mathbb{Z}$
let integerFromBitSeq bs =
  let (BitSeq len s bl) = cleanBitSeq bs in
 integerFromBoolList (s, bl)


assert integerFromBitSeq_0 : (integerFromBitSeq (BitSeq Nothing false [true; false; true]) = 5)
assert integerFromBitSeq_1 : (integerFromBitSeq (BitSeq (Just 2) false [true]) = 1)
assert integerFromBitSeq_2 : (integerFromBitSeq (BitSeq Nothing true [true; true; false]) = (−5))
assert integerFromBitSeq_3 : (integerFromBitSeq (BitSeq (Just 2) true [true; true; false]) = (−1))

lemma integerFromBitSeq_inv : (∀ i. integerFromBitSeq (bitSeqFromInteger Nothing i) = i)
assert integerFromBitSeq_inv_0 : (integerFromBitSeq (bitSeqFromInteger Nothing 10)) = 10
assert integerFromBitSeq_inv_1 : (integerFromBitSeq (bitSeqFromInteger Nothing (−1932))) = (−1932)
assert integerFromBitSeq_inv_2 : (integerFromBitSeq (bitSeqFromInteger Nothing 343)) = 343


(* Now we can via translation to integers map arithmetic operations to bitSequences *)

val bitSeqArithUnaryOp : ($\mathbb{Z}$ → $\mathbb{Z}$) → bitSequence → bitSequence
let bitSeqArithUnaryOp uop bs =
  let (BitSeq len _ _) = bs in
 bitSeqFromInteger len (uop (integerFromBitSeq bs))

val bitSeqArithBinOp : ($\mathbb{Z}$ → $\mathbb{Z}$ → $\mathbb{Z}$) → bitSequence →
bitSequence → bitSequence
let bitSeqArithBinOp binop bs_1 bs_2 =
  let (BitSeq len_1 _ _) = bs_1 in
 let (BitSeq len_2 _ _) = bs_2 in
 let len = match (len_1, len_2) with
   | (Just l_1, Just l_2) → Just (max l_1 l_2)
   | _ → Nothing
 end in
 bitSeqFromInteger len (binop (integerFromBitSeq bs_1) (integerFromBitSeq bs_2))

val bitSeqArithBinTest : ∀ α. ($\mathbb{Z}$ → $\mathbb{Z}$ → α) → bitSequence →
bitSequence → α
let bitSeqArithBinTest binop bs_1 bs_2 = binop (integerFromBitSeq bs_1) (integerFromBitSeq bs_2)


(* now instantiate the number interface for bit − sequences *)

val bitSeqFromNumeral : numeral → bitSequence
let inline bitSeqFromNumeral n = bitSeqFromInteger Nothing (integerFromNumeral n)
```

```
instance (Numeral BITSEQUENCE)
  let fromNumeral n =  bitSeqFromNumeral n
end

val bitSeqLess : BITSEQUENCE → BITSEQUENCE → BOOL
let bitSeqLess bs1 bs2 =  bitSeqArithBinTest (<) bs1 bs2

val bitSeqLessEqual : BITSEQUENCE → BITSEQUENCE → BOOL
let bitSeqLessEqual bs1 bs2 =  bitSeqArithBinTest (≤) bs1 bs2

val bitSeqGreater : BITSEQUENCE → BITSEQUENCE → BOOL
let bitSeqGreater bs1 bs2 =  bitSeqArithBinTest (>) bs1 bs2

val bitSeqGreaterEqual : BITSEQUENCE → BITSEQUENCE → BOOL
let bitSeqGreaterEqual bs1 bs2 =  bitSeqArithBinTest (≥) bs1 bs2

val bitSeqCompare : BITSEQUENCE → BITSEQUENCE → ORDERING
let bitSeqCompare bs1 bs2 =  bitSeqArithBinTest compare bs1 bs2

instance (Ord BITSEQUENCE)
  let compare =  bitSeqCompare
  let < =  bitSeqLess
  let <= =  bitSeqLessEqual
  let > =  bitSeqGreater
  let >= =  bitSeqGreaterEqual
end

instance (SetType BITSEQUENCE)
  let setElemCompare =  bitSeqCompare
end

(* arithmetic negation, don't mix up with bitwise negation *)
val bitSeqNegate : BITSEQUENCE → BITSEQUENCE
let bitSeqNegate bs =  bitSeqArithUnaryOp integerNegate bs

instance (NumNegate BITSEQUENCE)
  let ~ =  bitSeqNegate
end


val bitSeqAdd : BITSEQUENCE → BITSEQUENCE → BITSEQUENCE
let bitSeqAdd bs1 bs2 =  bitSeqArithBinOp (+) bs1 bs2

instance (NumAdd BITSEQUENCE)
  let + =  bitSeqAdd
end

val bitSeqMinus : BITSEQUENCE → BITSEQUENCE → BITSEQUENCE
let bitSeqMinus bs1 bs2 =  bitSeqArithBinOp (−) bs1 bs2

instance (NumMinus BITSEQUENCE)
  let − =  bitSeqMinus
end

val bitSeqSucc : BITSEQUENCE → BITSEQUENCE
let bitSeqSucc bs =  bitSeqArithUnaryOp succ bs

instance (NumSucc BITSEQUENCE)
```

```
  let succ =  bitSeqSucc
end

val bitSeqPred : BITSEQUENCE → BITSEQUENCE
let bitSeqPred bs =  bitSeqArithUnaryOp pred bs

instance (NumPred BITSEQUENCE)
  let pred =  bitSeqPred
end

val bitSeqMult : BITSEQUENCE → BITSEQUENCE → BITSEQUENCE
let bitSeqMult bs1 bs2 =  bitSeqArithBinOp integerMult bs1 bs2

instance (NumMult BITSEQUENCE)
  let * =  bitSeqMult
end


val bitSeqPow : BITSEQUENCE → NAT → BITSEQUENCE
let bitSeqPow bs n =  bitSeqArithUnaryOp (fun i → integerPow i n) bs

instance ( NumPow BITSEQUENCE )
  let ** =  bitSeqPow
end

val bitSeqDiv : BITSEQUENCE → BITSEQUENCE → BITSEQUENCE
let bitSeqDiv bs1 bs2 =  bitSeqArithBinOp integerDiv bs1 bs2

instance ( NumIntegerDivision BITSEQUENCE )
  let div =  bitSeqDiv
end

instance ( NumDivision BITSEQUENCE )
  let / =  bitSeqDiv
end

val bitSeqMod : BITSEQUENCE → BITSEQUENCE → BITSEQUENCE
let bitSeqMod bs1 bs2 =  bitSeqArithBinOp integerMod bs1 bs2

instance ( NumRemainder BITSEQUENCE )
  let mod =  bitSeqMod
end

val bitSeqMin : BITSEQUENCE → BITSEQUENCE → BITSEQUENCE
let bitSeqMin bs1 bs2 =  bitSeqArithBinOp integerMin bs1 bs2

val bitSeqMax : BITSEQUENCE → BITSEQUENCE → BITSEQUENCE
let bitSeqMax bs1 bs2 =  bitSeqArithBinOp integerMax bs1 bs2

instance ( OrdMaxMin BITSEQUENCE )
  let max =  bitSeqMax
  let min =  bitSeqMin
end

assert bitSequence_test1 : (2 + (5 : BITSEQUENCE) = 7)
assert bitSequence_test2 : (8 − (7 : BITSEQUENCE) = 1)
assert bitSequence_test3 : (7 − (8 : BITSEQUENCE) = −1)
assert bitSequence_test4 : (7 ∗ (8 : BITSEQUENCE) = 56)
```

assert *bitSequence_test*$_5$: $((7 : \text{BITSEQUENCE})^2 = 49)$
assert *bitSequence_test*$_6$: $(\text{div } 11\ (4 : \text{BITSEQUENCE}) = 2)$
assert *bitSequence_test6a* : $(\text{div } (-11)\ (4 : \text{BITSEQUENCE}) = -3)$
assert *bitSequence_test*$_7$: $(11 / (4 : \text{BITSEQUENCE}) = 2)$
assert *bitSequence_test7a* : $(-11 / (4 : \text{BITSEQUENCE}) = -3)$
assert *bitSequence_test*$_8$: $(11 \bmod (4 : \text{BITSEQUENCE}) = 3)$
assert *bitSequence_test8a* : $(-11 \bmod (4 : \text{BITSEQUENCE}) = 1)$
assert *bitSequence_test*$_9$: $(11 < (12 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{10}$: $(11 \leq (12 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{11}$: $(12 \leq (12 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{12}$: $(\neg\ (12 < (12 : \text{BITSEQUENCE})))$
assert *bitSequence_test*$_{13}$: $(12 > (11 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{14}$: $(12 \geq (11 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{15}$: $(12 \geq (12 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{16}$: $(\neg\ (12 > (12 : \text{BITSEQUENCE})))$
assert *bitSequence_test*$_{17}$: $(\min\ 12\ (12 : \text{BITSEQUENCE}) = 12)$
assert *bitSequence_test*$_{18}$: $(\min\ 10\ (12 : \text{BITSEQUENCE}) = 10)$
assert *bitSequence_test*$_{19}$: $(\min\ 12\ (10 : \text{BITSEQUENCE}) = 10)$
assert *bitSequence_test*$_{20}$: $(\max\ 12\ (12 : \text{BITSEQUENCE}) = 12)$
assert *bitSequence_test*$_{21}$: $(\max\ 10\ (12 : \text{BITSEQUENCE}) = 12)$
assert *bitSequence_test*$_{22}$: $(\max\ 12\ (10 : \text{BITSEQUENCE}) = 12)$
assert *bitSequence_test*$_{23}$: $(\text{succ } 12 = (13 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{24}$: $(\text{succ } 0 = (1 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{25}$: $(\text{pred } 12 = (11 : \text{BITSEQUENCE}))$
assert *bitSequence_test*$_{26}$: $(\text{pred } 0 = -(1 : \text{BITSEQUENCE}))$

(* ========== *)
(* Interface for bitoperations *)
(* ========== *)

class (*WordNot* α)
  val *lnot* : $\alpha \to \alpha$
end

class (*WordAnd* α)
  val *land* [conjunction] : $\alpha \to \alpha \to \alpha$
end

class (*WordOr* α)
  val *lor* [inclusive_or] : $\alpha \to \alpha \to \alpha$
end

class (*WordXor* α)
  val *lxor* [exclusive_or] : $\alpha \to \alpha \to \alpha$
end

class (*WordLsl* α)
  val *lsl* [left_shift] : $\alpha \to \text{NAT} \to \alpha$
end

class (*WordLsr* α)
  val *lsr* [logicial_right_shift] : $\alpha \to \text{NAT} \to \alpha$
end

```
class ( WordAsr α )
  val asr [arithmetic_right_shift] : α → NAT → α
end

(* ------------------------- *)
(* bitSequence          *)
(* ------------------------- *)

instance ( WordNot BITSEQUENCE)
  let lnot = bitSeqNot
end

instance ( WordAnd BITSEQUENCE)
  let land = bitSeqAnd
end

instance ( WordOr BITSEQUENCE)
  let lor = bitSeqOr
end

instance ( WordXor BITSEQUENCE)
  let lxor = bitSeqXor
end

instance (WordLsl BITSEQUENCE)
  let lsl = bitSeqShiftLeft
end

instance (WordLsr BITSEQUENCE)
  let lsr = bitSeqLogicalShiftRight
end

instance (WordAsr BITSEQUENCE)
  let asr = bitSeqArithmeticShiftRight
end

assert bitSequence_bittest_1 : ((6 : BITSEQUENCE) land 5 = 4)
assert bitSequence_bittest_2 : ((6 : BITSEQUENCE) lor 5 = 7)
assert bitSequence_bittest_3 : ((6 : BITSEQUENCE) lxor 5 = 3)
assert bitSequence_bittest_4 : ((12 : BITSEQUENCE) land 9 = 8)
assert bitSequence_bittest_5 : ((12 : BITSEQUENCE) lor 9 = 13)
assert bitSequence_bittest_6 : ((12 : BITSEQUENCE) lxor 9 = 5)

assert bitSequence_bittest_7 : (lnot (12 : BITSEQUENCE) = −13)
assert bitSequence_bittest_8 : (lnot (27 : BITSEQUENCE) = −28)
assert bitSequence_bittest_9 : ((27 : BITSEQUENCE) lsl 0 = 27)
assert bitSequence_bittest_10 : ((27 : BITSEQUENCE) lsl 1 = 54)
assert bitSequence_bittest_11 : ((27 : BITSEQUENCE) lsl 2 = 108)
assert bitSequence_bittest_12 : ((27 : BITSEQUENCE) lsl 3 = 216)
assert bitSequence_bittest_13 : ((27 : BITSEQUENCE) lsr 0 = 27)
assert bitSequence_bittest_14 : ((27 : BITSEQUENCE) lsr 1 = 13)
assert bitSequence_bittest_15 : ((27 : BITSEQUENCE) lsr 2 = 6)
assert bitSequence_bittest_16 : ((27 : BITSEQUENCE) lsr 3 = 3)
assert bitSequence_bittest_17 : ((27 : BITSEQUENCE) asr 0 = 27)
assert bitSequence_bittest_18 : ((27 : BITSEQUENCE) asr 1 = 13)
assert bitSequence_bittest_19 : ((27 : BITSEQUENCE) asr 2 = 6)
assert bitSequence_bittest_20 : ((27 : BITSEQUENCE) asr 3 = 3)
assert bitSequence_bittest_21 : ((−(27 : BITSEQUENCE)) lsr 0 = −(27))
```

assert *bitSequence_bittest*$_{22}$: $((-(27$: BITSEQUENCE) asr 0) = $-(27))$
assert *bitSequence_bittest*$_{23}$: $((-(27$: BITSEQUENCE)) lsr 1 = $-(14))$
assert *bitSequence_bittest*$_{24}$: $((-(27$: BITSEQUENCE)) asr 1 = $-(14))$

(∗ −−−−−−−−−−−−−−−−−−−−−−−− ∗)
(∗ int32 ∗)
(∗ −−−−−−−−−−−−−−−−−−−−−−−− ∗)

val *int32Lnot* : INT$_{32}$ → INT$_{32}$
declare *ocaml* target_rep function int32Lnot = 'Int32.lognot'
declare *hol* target_rep function int32Lnot *w* = ('~' *w*)
declare *isabelle* target_rep function int32Lnot *w* = ('NOT' *w*)
declare *coq* target_rep function int32Lnot *w* = *w* (∗ XXX : fix ∗)

instance (*WordNot* INT$_{32}$)
  let *lnot* = int32Lnot
end

val *int32Lor* : INT$_{32}$ → INT$_{32}$ → INT$_{32}$
declare *ocaml* target_rep function int32Lor = 'Int32.logor'
declare *hol* target_rep function int32Lor = 'word_or'
declare *isabelle* target_rep function int32Lor = infix 'OR'
declare *coq* target_rep function int32Lor *q* *w* = *w* (∗ XXX : fix ∗)

instance (*WordOr* INT$_{32}$)
  let *lor* = int32Lor
end

val *int32Lxor* : INT$_{32}$ → INT$_{32}$ → INT$_{32}$
declare *ocaml* target_rep function int32Lxor = 'Int32.logxor'
declare *hol* target_rep function int32Lxor = 'word_xor'
declare *isabelle* target_rep function int32Lxor = infix 'XOR'
declare *coq* target_rep function int32Lxor *q* *w* = *w* (∗ XXX : fix ∗)

instance (*WordXor* INT$_{32}$)
  let *lxor* = int32Lxor
end

val *int32Land* : INT$_{32}$ → INT$_{32}$ → INT$_{32}$
declare *ocaml* target_rep function int32Land = 'Int32.logand'
declare *hol* target_rep function int32Land = 'word_and'
declare *isabelle* target_rep function int32Land = infix 'AND'
declare *coq* target_rep function int32Land *q* *w* = *w* (∗ XXX : fix ∗)

instance (*WordAnd* INT$_{32}$)
  let *land* = int32Land
end

val *int32Lsl* : INT$_{32}$ → NAT → INT$_{32}$
declare *ocaml* target_rep function int32Lsl = 'Int32.shift_left'
declare *hol* target_rep function int32Lsl = 'word_lsl'
declare *isabelle* target_rep function int32Lsl = infix '<<'
declare *coq* target_rep function int32Lsl *q* *w* = *q* (∗ XXX : fix ∗)

instance (*WordLsl* INT$_{32}$)
  let *lsl* = int32Lsl

```
end

val int32Lsr : INT32 → NAT → INT32
declare ocaml target_rep function int32Lsr = 'Int32.shift_right_logical'
declare hol target_rep function int32Lsr = 'word_lsr'
declare isabelle target_rep function int32Lsr = infix '>>'
declare coq target_rep function int32Lsr q w = q (* XXX: fix *)

instance (WordLsr INT32)
  let lsr = int32Lsr
end


val int32Asr : INT32 → NAT → INT32
declare ocaml target_rep function int32Asr = 'Int32.shift_right'
declare hol target_rep function int32Asr = 'word_asr'
declare isabelle target_rep function int32Asr = infix '>>>'
declare coq target_rep function int32Asr q w = q (* XXX: fix *)

instance (WordAsr INT32)
  let asr = int32Asr
end


assert int32_bittest1 : ((6 : INT32) land 5 = 4)
assert int32_bittest2 : ((6 : INT32) lor 5 = 7)
assert int32_bittest3 : ((6 : INT32) lxor 5 = 3)
assert int32_bittest4 : ((12 : INT32) land 9 = 8)
assert int32_bittest5 : ((12 : INT32) lor 9 = 13)
assert int32_bittest6 : ((12 : INT32) lxor 9 = 5)

assert int32_bittest7 : (lnot (12 : INT32) = −13)
assert int32_bittest8 : (lnot (27 : INT32) = −28)
assert int32_bittest9 : ((27 : INT32) lsl 0 = 27)
assert int32_bittest10 : ((27 : INT32) lsl 1 = 54)
assert int32_bittest11 : ((27 : INT32) lsl 2 = 108)
assert int32_bittest12 : ((27 : INT32) lsl 3 = 216)
assert int32_bittest13 : ((27 : INT32) lsr 0 = 27)
assert int32_bittest14 : ((27 : INT32) lsr 1 = 13)
assert int32_bittest15 : ((27 : INT32) lsr 2 = 6)
assert int32_bittest16 : ((27 : INT32) lsr 3 = 3)
assert int32_bittest17 : ((27 : INT32) asr 0 = 27)
assert int32_bittest18 : ((27 : INT32) asr 1 = 13)
assert int32_bittest19 : ((27 : INT32) asr 2 = 6)
assert int32_bittest20 : ((27 : INT32) asr 3 = 3)
assert int32_bittest21 : ((−(27 : INT32)) lsr 0 = −(27))
assert int32_bittest22 : ((−(27 : INT32) asr 0) = −(27))
assert int32_bittest23 : ((−(27 : INT32)) lsr 2 = 1073741817)
assert int32_bittest24 : ((−(27 : INT32)) asr 2 = −(7))


(* ――――――――――――――――――――――――― *)
(* int64                *)
(* ――――――――――――――――――――――――― *)

val int64Lnot : INT64 → INT64
declare ocaml target_rep function int64Lnot = 'Int64.lognot'
declare hol target_rep function int64Lnot w = ('~' w)
```

```
declare isabelle target_rep function int64Lnot w = ('NOT' w)
declare coq target_rep function int64Lnot w = w (* XXX: fix *)

instance ( WordNot INT64)
  let lnot = int64Lnot
end

val int64Lor : INT64 → INT64 → INT64
declare ocaml target_rep function int64Lor = 'Int64.logor'
declare hol target_rep function int64Lor = 'word_or'
declare isabelle target_rep function int64Lor = infix 'OR'
declare coq target_rep function int64Lor q w = w (* XXX: fix *)

instance (WordOr INT64)
  let lor = int64Lor
end

val int64Lxor : INT64 → INT64 → INT64
declare ocaml target_rep function int64Lxor = 'Int64.logxor'
declare hol target_rep function int64Lxor = 'word_xor'
declare isabelle target_rep function int64Lxor = infix 'XOR'
declare coq target_rep function int64Lxor q w = w (* XXX: fix *)

instance (WordXor INT64)
  let lxor = int64Lxor
end

val int64Land : INT64 → INT64 → INT64
declare ocaml target_rep function int64Land = 'Int64.logand'
declare hol target_rep function int64Land = 'word_and'
declare isabelle target_rep function int64Land = infix 'AND'
declare coq target_rep function int64Land q w = w (* XXX: fix *)

instance (WordAnd INT64)
  let land = int64Land
end

val int64Lsl : INT64 → NAT → INT64
declare ocaml target_rep function int64Lsl = 'Int64.shift_left'
declare hol target_rep function int64Lsl = 'word_lsl'
declare isabelle target_rep function int64Lsl = infix '<<'
declare coq target_rep function int64Lsl q w = q (* XXX: fix *)

instance (WordLsl INT64)
  let lsl = int64Lsl
end

val int64Lsr : INT64 → NAT → INT64
declare ocaml target_rep function int64Lsr = 'Int64.shift_right_logical'
declare hol target_rep function int64Lsr = 'word_lsr'
declare isabelle target_rep function int64Lsr = infix '>>'
declare coq target_rep function int64Lsr q w = q (* XXX: fix *)

instance (WordLsr INT64)
  let lsr = int64Lsr
end

val int64Asr : INT64 → NAT → INT64
```

```
declare ocaml target_rep function int64Asr = 'Int64.shift_right'
declare hol target_rep function int64Asr = 'word_asr'
declare isabelle target_rep function int64Asr = infix '>>>'
declare coq target_rep function int64Asr q w = q (* XXX: fix *)

instance (WordAsr INT64)
  let asr = int64Asr
end


assert int64_bittest_1 : ((6 : INT64) land 5 = 4)
assert int64_bittest_2 : ((6 : INT64) lor 5 = 7)
assert int64_bittest_3 : ((6 : INT64) lxor 5 = 3)
assert int64_bittest_4 : ((12 : INT64) land 9 = 8)
assert int64_bittest_5 : ((12 : INT64) lor 9 = 13)
assert int64_bittest_6 : ((12 : INT64) lxor 9 = 5)

assert int64_bittest_7 : (lnot (12 : INT64) = -13)
assert int64_bittest_8 : (lnot (27 : INT64) = -28)
assert int64_bittest_9 : ((27 : INT64) lsl 0 = 27)
assert int64_bittest_10 : ((27 : INT64) lsl 1 = 54)
assert int64_bittest_11 : ((27 : INT64) lsl 2 = 108)
assert int64_bittest_12 : ((27 : INT64) lsl 3 = 216)
assert int64_bittest_13 : ((27 : INT64) lsr 0 = 27)
assert int64_bittest_14 : ((27 : INT64) lsr 1 = 13)
assert int64_bittest_15 : ((27 : INT64) lsr 2 = 6)
assert int64_bittest_16 : ((27 : INT64) lsr 3 = 3)
assert int64_bittest_17 : ((27 : INT64) asr 0 = 27)
assert int64_bittest_18 : ((27 : INT64) asr 1 = 13)
assert int64_bittest_19 : ((27 : INT64) asr 2 = 6)
assert int64_bittest_20 : ((27 : INT64) asr 3 = 3)
assert int64_bittest_21 : ((-(27 : INT64)) lsr 0 = -(27))
assert int64_bittest_22 : ((-(27 : INT64) asr 0) = -(27))
assert int64_bittest_23 : ((-(27 : INT64)) lsr 34 = 1073741823)
assert int64_bittest_24 : ((-(27 : INT64)) asr 2 = -(7))


(* ------------------------------------------------- *)
(* Words via bit sequences *)
(* ------------------------------------------------- *)

val defaultLnot : ∀ α. (BITSEQUENCE → α) → (α → BITSEQUENCE) → α → α
let defaultLnot fromBitSeq toBitSeq x = fromBitSeq (bitSeqNegate (toBitSeq x))

val defaultLand : ∀ α. (BITSEQUENCE → α) → (α → BITSEQUENCE) → α → α → α
let defaultLand fromBitSeq toBitSeq x1 x2 = fromBitSeq (bitSeqAnd (toBitSeq x1) (toBitSeq x2))

val defaultLor : ∀ α. (BITSEQUENCE → α) → (α → BITSEQUENCE) → α → α → α
let defaultLor fromBitSeq toBitSeq x1 x2 = fromBitSeq (bitSeqOr (toBitSeq x1) (toBitSeq x2))

val defaultLxor : ∀ α. (BITSEQUENCE → α) → (α → BITSEQUENCE) → α → α → α
let defaultLxor fromBitSeq toBitSeq x1 x2 = fromBitSeq (bitSeqXor (toBitSeq x1) (toBitSeq x2))

val defaultLsl : ∀ α. (BITSEQUENCE → α) → (α → BITSEQUENCE) → α → NAT → α
let defaultLsl fromBitSeq toBitSeq x n = fromBitSeq (bitSeqShiftLeft (toBitSeq x) n)

val defaultLsr : ∀ α. (BITSEQUENCE → α) → (α → BITSEQUENCE) → α → NAT → α
let defaultLsr fromBitSeq toBitSeq x n = fromBitSeq (bitSeqLogicalShiftRight (toBitSeq x) n)
```

```
val defaultAsr : ∀ α. (bitSequence → α) → (α → bitSequence) → α → nat → α
let defaultAsr fromBitSeq toBitSeq x n =  fromBitSeq (bitSeqArithmeticShiftRight (toBitSeq x) n)

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)
(* integer                                                *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

val integerLnot : $\mathbb{Z}$ → $\mathbb{Z}$
let integerLnot i =  −(i + 1)

instance ( WordNot $\mathbb{Z}$)
  let lnot =  integerLnot
end


val integerLor : $\mathbb{Z}$ → $\mathbb{Z}$ → $\mathbb{Z}$
let integerLor i1 i2 =  defaultLor integerFromBitSeq (bitSeqFromInteger Nothing) i1 i2
declare ocaml target_rep function integerLor = 'Nat_big_num.bitwise_or'

instance (WordOr $\mathbb{Z}$)
  let lor =  integerLor
end

val integerLxor : $\mathbb{Z}$ → $\mathbb{Z}$ → $\mathbb{Z}$
let integerLxor i1 i2 =  defaultLxor integerFromBitSeq (bitSeqFromInteger Nothing) i1 i2
declare ocaml target_rep function integerLxor = 'Nat_big_num.bitwise_xor'

instance (WordXor $\mathbb{Z}$)
  let lxor =  integerLxor
end

val integerLand : $\mathbb{Z}$ → $\mathbb{Z}$ → $\mathbb{Z}$
let integerLand i1 i2 =  defaultLand integerFromBitSeq (bitSeqFromInteger Nothing) i1 i2
declare ocaml target_rep function integerLand = 'Nat_big_num.bitwise_and'

instance (WordAnd $\mathbb{Z}$)
  let land =  integerLand
end

val integerLsl : $\mathbb{Z}$ → nat → $\mathbb{Z}$
let integerLsl i n =  defaultLsl integerFromBitSeq (bitSeqFromInteger Nothing) i n
declare ocaml target_rep function integerLsl = 'Nat_big_num.shift_left'

instance (WordLsl $\mathbb{Z}$)
  let lsl =  integerLsl
end

val integerAsr : $\mathbb{Z}$ → nat → $\mathbb{Z}$
let integerAsr i n =  defaultAsr integerFromBitSeq (bitSeqFromInteger Nothing) i n
declare ocaml target_rep function integerAsr = 'Nat_big_num.shift_right'

instance (WordLsr $\mathbb{Z}$)
  let lsr =  integerAsr
end

instance (WordAsr $\mathbb{Z}$)
  let asr =  integerAsr
```

```
end



assert integer_bittest1 : ((6 : $\MATHBB{Z}$) land 5 = 4)
assert integer_bittest2 : ((6 : $\MATHBB{Z}$) lor 5 = 7)
assert integer_bittest3 : ((6 : $\MATHBB{Z}$) lxor 5 = 3)
assert integer_bittest4 : ((12 : $\MATHBB{Z}$) land 9 = 8)
assert integer_bittest5 : ((12 : $\MATHBB{Z}$) lor 9 = 13)
assert integer_bittest6 : ((12 : $\MATHBB{Z}$) lxor 9 = 5)

assert integer_bittest7 : (lnot (12 : $\MATHBB{Z}$) = −13)
assert integer_bittest8 : (lnot (27 : $\MATHBB{Z}$) = −28)
assert integer_bittest9 : ((27 : $\MATHBB{Z}$) lsl 0 = 27)
assert integer_bittest10 : ((27 : $\MATHBB{Z}$) lsl 1 = 54)
assert integer_bittest11 : ((27 : $\MATHBB{Z}$) lsl 2 = 108)
assert integer_bittest12 : ((27 : $\MATHBB{Z}$) lsl 3 = 216)
assert integer_bittest13 : ((27 : $\MATHBB{Z}$) lsr 0 = 27)
assert integer_bittest14 : ((27 : $\MATHBB{Z}$) lsr 1 = 13)
assert integer_bittest15 : ((27 : $\MATHBB{Z}$) lsr 2 = 6)
assert integer_bittest16 : ((27 : $\MATHBB{Z}$) lsr 3 = 3)
assert integer_bittest17 : ((27 : $\MATHBB{Z}$) asr 0 = 27)
assert integer_bittest18 : ((27 : $\MATHBB{Z}$) asr 1 = 13)
assert integer_bittest19 : ((27 : $\MATHBB{Z}$) asr 2 = 6)
assert integer_bittest20 : ((27 : $\MATHBB{Z}$) asr 3 = 3)
assert integer_bittest22 : ((−(27 : $\MATHBB{Z}$) asr 0) = −(27))
assert integer_bittest24 : ((−(27 : $\MATHBB{Z}$)) asr 2 = −(7))


(* − − − − − − − − − − − − − − − − − − − − − − − − − *)
(* int            *)
(* − − − − − − − − − − − − − − − − − − − − − − − − − *)

(* sometimes it is convenient to be able to perform bit − operations on ints.  However, since int is not well

val intFromBitSeq : BITSEQUENCE → INT
let intFromBitSeq bs =  intFromInteger (integerFromBitSeq (resizeBitSeq (Just 31) bs))


val bitSeqFromInt : INT → BITSEQUENCE
let bitSeqFromInt i =  bitSeqFromInteger (Just 31) (integerFromInt i)


val intLnot : INT → INT
let intLnot i =  −(i + 1)
declare ocaml target_rep function intLnot = 'lnot'

instance ( WordNot INT)
  let lnot =  intLnot
end

val intLor : INT → INT → INT
let intLor i1 i2 =  defaultLor intFromBitSeq bitSeqFromInt i1 i2
declare ocaml target_rep function intLor = infix 'lor'

instance (WordOr INT)
  let lor =  intLor
end
```

val $intLxor$: INT → INT → INT
let $intLxor$ i_1 i_2 = defaultLxor intFromBitSeq bitSeqFromInt i_1 i_2
declare $ocaml$ target_rep function intLxor = infix 'lxor'

instance ($WordXor$ INT)
  let $lxor$ = intLxor
end

val $intLand$: INT → INT → INT
let $intLand$ i_1 i_2 = defaultLand intFromBitSeq bitSeqFromInt i_1 i_2
declare $ocaml$ target_rep function intLand = infix 'land'

instance ($WordAnd$ INT)
  let $land$ = intLand
end

val $intLsl$: INT → NAT → INT
let $intLsl$ i n = defaultLsl intFromBitSeq bitSeqFromInt i n
declare $ocaml$ target_rep function intLsl = infix 'lsl'

instance ($WordLsl$ INT)
  let lsl = intLsl
end

val $intAsr$: INT → NAT → INT
let $intAsr$ i n = defaultAsr intFromBitSeq bitSeqFromInt i n
declare $ocaml$ target_rep function intAsr = infix 'asr'

instance ($WordAsr$ INT)
  let asr = intAsr
end

assert $int\_bittest_1$: ((6 : INT) land 5 = 4)
assert $int\_bittest_2$: ((6 : INT) lor 5 = 7)
assert $int\_bittest_3$: ((6 : INT) lxor 5 = 3)
assert $int\_bittest_4$: ((12 : INT) land 9 = 8)
assert $int\_bittest_5$: ((12 : INT) lor 9 = 13)
assert $int\_bittest_6$: ((12 : INT) lxor 9 = 5)

assert $int\_bittest_7$: (lnot (12 : INT) = −13)
assert $int\_bittest_8$: (lnot (27 : INT) = −28)
assert $int\_bittest_9$: ((27 : INT) lsl 0 = 27)
assert $int\_bittest_{10}$: ((27 : INT) lsl 1 = 54)
assert $int\_bittest_{11}$: ((27 : INT) lsl 2 = 108)
assert $int\_bittest_{12}$: ((27 : INT) lsl 3 = 216)
assert $int\_bittest_{17}$: ((27 : INT) asr 0 = 27)
assert $int\_bittest_{18}$: ((27 : INT) asr 1 = 13)
assert $int\_bittest_{19}$: ((27 : INT) asr 2 = 6)
assert $int\_bittest_{20}$: ((27 : INT) asr 3 = 3)
assert $int\_bittest_{22}$: ((−(27 : INT) asr 0) = −(27))
assert $int\_bittest_{24}$: ((−(27 : INT)) asr 2 = −(7))

(∗ − ∗)

```
(* natural            *)
(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

(* some operations work also on positive numbers *)

val naturalFromBitSeq : bitSequence → $\mathbb{N}$
let naturalFromBitSeq bs =  naturalFromInteger (integerFromBitSeq bs)

val bitSeqFromNatural : maybe nat → $\mathbb{N}$ → bitSequence
let bitSeqFromNatural len n =  bitSeqFromInteger len (integerFromNatural n)

val naturalLor : $\mathbb{N}$ → $\mathbb{N}$ → $\mathbb{N}$
let naturalLor i1 i2 =  defaultLor naturalFromBitSeq (bitSeqFromNatural Nothing) i1 i2
declare ocaml target_rep function naturalLor = 'Nat_big_num.bitwise_or'

instance (WordOr $\mathbb{N}$)
  let lor =  naturalLor
end

val naturalLxor : $\mathbb{N}$ → $\mathbb{N}$ → $\mathbb{N}$
let naturalLxor i1 i2 =  defaultLxor naturalFromBitSeq (bitSeqFromNatural Nothing) i1 i2
declare ocaml target_rep function naturalLxor = 'Nat_big_num.bitwise_xor'

instance (WordXor $\mathbb{N}$)
  let lxor =  naturalLxor
end

val naturalLand : $\mathbb{N}$ → $\mathbb{N}$ → $\mathbb{N}$
let naturalLand i1 i2 =  defaultLand naturalFromBitSeq (bitSeqFromNatural Nothing) i1 i2
declare ocaml target_rep function naturalLand = 'Nat_big_num.bitwise_and'

instance (WordAnd $\mathbb{N}$)
  let land =  naturalLand
end

val naturalLsl : $\mathbb{N}$ → nat → $\mathbb{N}$
let naturalLsl i n =  defaultLsl naturalFromBitSeq (bitSeqFromNatural Nothing) i n
declare ocaml target_rep function naturalLsl = 'Nat_big_num.shift_left'

instance (WordLsl $\mathbb{N}$)
  let lsl =  naturalLsl
end

val naturalAsr : $\mathbb{N}$ → nat → $\mathbb{N}$
let naturalAsr i n =  defaultAsr naturalFromBitSeq (bitSeqFromNatural Nothing) i n
declare ocaml target_rep function naturalAsr = 'Nat_big_num.shift_right'

instance (WordLsr $\mathbb{N}$)
  let lsr =  naturalAsr
end

instance (WordAsr $\mathbb{N}$)
  let asr =  naturalAsr
end



assert natural_bittest1 : ((6 : $\mathbb{N}$) land 5 = 4)
```

```
assert natural_bittest2 : ((6 : $\mathbb{N}$) lor 5 = 7)
assert natural_bittest3 : ((6 : $\mathbb{N}$) lxor 5 = 3)
assert natural_bittest4 : ((12 : $\mathbb{N}$) land 9 = 8)
assert natural_bittest5 : ((12 : $\mathbb{N}$) lor 9 = 13)
assert natural_bittest6 : ((12 : $\mathbb{N}$) lxor 9 = 5)

assert natural_bittest9 : ((27 : $\mathbb{N}$) lsl 0 = 27)
assert natural_bittest10 : ((27 : $\mathbb{N}$) lsl 1 = 54)
assert natural_bittest11 : ((27 : $\mathbb{N}$) lsl 2 = 108)
assert natural_bittest12 : ((27 : $\mathbb{N}$) lsl 3 = 216)
assert natural_bittest13 : ((27 : $\mathbb{N}$) lsr 0 = 27)
assert natural_bittest14 : ((27 : $\mathbb{N}$) lsr 1 = 13)
assert natural_bittest15 : ((27 : $\mathbb{N}$) lsr 2 = 6)
assert natural_bittest16 : ((27 : $\mathbb{N}$) lsr 3 = 3)
assert natural_bittest17 : ((27 : $\mathbb{N}$) asr 0 = 27)
assert natural_bittest18 : ((27 : $\mathbb{N}$) asr 1 = 13)
assert natural_bittest19 : ((27 : $\mathbb{N}$) asr 2 = 6)
assert natural_bittest20 : ((27 : $\mathbb{N}$) asr 3 = 3)


(* −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−− *)
(* nat                *)
(* −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−− *)

(* sometimes it is convenient to be able to perform bit − operations on nats.  However, since nat is not well

val natFromBitSeq : BITSEQUENCE → NAT
let natFromBitSeq bs = natFromNatural (naturalFromBitSeq (resizeBitSeq (Just 31) bs))


val bitSeqFromNat : NAT → BITSEQUENCE
let bitSeqFromNat i = bitSeqFromNatural (Just 31) (naturalFromNat i)


val natLor : NAT → NAT → NAT
let natLor i1 i2 = defaultLor natFromBitSeq bitSeqFromNat i1 i2
declare ocaml target_rep function natLor = infix 'lor'

instance (WordOr NAT)
  let lor = natLor
end

val natLxor : NAT → NAT → NAT
let natLxor i1 i2 = defaultLxor natFromBitSeq bitSeqFromNat i1 i2
declare ocaml target_rep function natLxor = infix 'lxor'

instance (WordXor NAT)
  let lxor = natLxor
end

val natLand : NAT → NAT → NAT
let natLand i1 i2 = defaultLand natFromBitSeq bitSeqFromNat i1 i2
declare ocaml target_rep function natLand = infix 'land'

instance (WordAnd NAT)
  let land = natLand
end
```

```
val natLsl : NAT → NAT → NAT
let natLsl i n = defaultLsl natFromBitSeq bitSeqFromNat i n
declare ocaml target_rep function natLsl = infix 'lsl'

instance (WordLsl NAT)
  let lsl = natLsl
end

val natAsr : NAT → NAT → NAT
let natAsr i n = defaultAsr natFromBitSeq bitSeqFromNat i n
declare ocaml target_rep function natAsr = infix 'asr'

instance (WordAsr NAT)
  let asr = natAsr
end



assert nat_bittest1 : ((6 : NAT) land 5 = 4)
assert nat_bittest2 : ((6 : NAT) lor 5 = 7)
assert nat_bittest3 : ((6 : NAT) lxor 5 = 3)
assert nat_bittest4 : ((12 : NAT) land 9 = 8)
assert nat_bittest5 : ((12 : NAT) lor 9 = 13)
assert nat_bittest6 : ((12 : NAT) lxor 9 = 5)

assert nat_bittest9 : ((27 : NAT) lsl 0 = 27)
assert nat_bittest10 : ((27 : NAT) lsl 1 = 54)
assert nat_bittest11 : ((27 : NAT) lsl 2 = 108)
assert nat_bittest12 : ((27 : NAT) lsl 3 = 216)
assert nat_bittest17 : ((27 : NAT) asr 0 = 27)
assert nat_bittest18 : ((27 : NAT) asr 1 = 13)
assert nat_bittest19 : ((27 : NAT) asr 2 = 6)
assert nat_bittest20 : ((27 : NAT) asr 3 = 3)
```

24 Show

```
declare {isabelle, hol, ocaml, coq} rename module = lem_show

open import String Maybe Num Basic_classes

open import {hol} lemTheory

class (Show α)
  val show : α → STRING
end

instance (Show STRING)
  let show s = ^ "\"" ^ s "\""
end

val stringFromMaybe : ∀ α. (α → STRING) → MAYBE α → STRING
let stringFromMaybe showX x =
   match x with
    | Just x → ^ "Just (" ^ showX x ")"
    | Nothing → "Nothing"
  end

instance ∀ α. Show α ⇒ (Show (MAYBE α))
  let show x_opt = stringFromMaybe show x_opt
end

val stringFromListAux : ∀ α. (α → STRING) → LIST α → STRING
let rec stringFromListAux showX x =
   match x with
    | [] → ""
    | x :: xs′ →
      match xs′ with
      | [] → showX x
      | _ → ^ showX x ^ "; " stringFromListAux showX xs′
      end
  end

val stringFromList : ∀ α. (α → STRING) → LIST α → STRING
let stringFromList showX xs =
   ^ "[" ^ stringFromListAux showX xs "]"

instance ∀ α. Show α ⇒ (Show (LIST α))
  let show xs = stringFromList show xs
end

val stringFromPair : ∀ α β. (α → STRING) → (β → STRING) → (α * β) → STRING
let stringFromPair showX showY (x, y) =
   ^ "(" ^ showX x ^ ", " ^ showY y ")"

instance ∀ α β. Show α, Show β ⇒ (Show (α * β))
  let show = stringFromPair show show
end

instance (Show BOOL)
  let show b = if b then "true" else "false"
end
```

25 Show_extra

```
declare {isabelle, hol, ocaml, coq} rename module = lem_show_extra

open import String Maybe Num Basic_classes Set Relation Show
import Set_extra String_extra

instance (Show NAT)
  let show = String_extra.stringFromNat
end

instance (Show $\MATHBB{N}$)
  let show = String_extra.stringFromNatural
end

instance (Show INT)
  let show = String_extra.stringFromInt
end

instance (Show $\MATHBB{Z}$)
  let show = String_extra.stringFromInteger
end

let stringFromSet showX xs =
   ^ "{" ^ Show.stringFromListAux showX (Set_extra.toList xs) "}"

(* Abbreviates the representation if the relation is transitive. *)
let stringFromRelation showX rel1 =
   if isTransitive rel1 then
    let pruned_rel = withoutTransitiveEdges rel1 in
    if (∀ e ∈ rel1. (e ∈ pruned_rel)) then
    (* The relations are the same (there are no transitive edges),       so we can just as well print the origin
      stringFromSet showX rel1
    else
      ^ "trancl of " stringFromSet showX pruned_rel
  else
    stringFromSet showX rel1

instance ∀ α. Show α, SetType α ⇒ (Show (SET α))
  let show xs = stringFromSet show xs
end
```

26 Machine_word

```
(*****************************************************************************)
(* A new machine word library, suitable for targetting from Sail,  *)
(* and a thin wrapper around the HOL and Isabelle word libraries.  *)
(*****************************************************************************)

declare {isabelle, coq, hol, ocaml} rename module = Lem_machine_word

open import Bool Num Basic_classes Show Function

open import {isabelle} HOL - Word.Word
open import {hol} wordsTheory wordsLib bitstringTheory integer_wordTheory

type MWORD α
declare isabelle target_sorts MWORD = len

class (Size α)
  val size : NAT
end

declare isabelle target_rep type MWORD α = 'Word.word' α
declare hol target_rep type MWORD α = 'words$word' α
declare ocaml target_rep type MWORD α = 'Lem.mword'

val native_size : ∀ α. NAT
declare hol target_rep function native_size = 'dimindex' ('the_value' : 'itself' α)
declare isabelle target_rep function native_size = 'len_of' ('TYPE(_)' : 'itself' α)

let inline {isabelle, hol} size = native_size

val ocaml_inject : ∀ α. NAT * $\MATHBB{N}$ → MWORD α
declare ocaml target_rep function ocaml_inject = 'Lem.machine_word_inject'

(* A singleton type family that can be used to carry a size as the type parameter *)

type ITSELF α
declare isabelle target_sorts ITSELF = len
declare hol target_rep type ITSELF α = 'itself' α
declare isabelle target_rep type ITSELF α = 'itself' α
declare ocaml target_rep type ITSELF α = 'unit'

val the_value : ∀ α. ITSELF α
declare hol target_rep function the_value = 'the_value'
declare isabelle target_rep function the_value = 'TYPE(_)'
declare ocaml target_rep function the_value = '()'

val size_itself : ∀ α. Size α ⇒ ITSELF α → NAT
let size_itself x = size

(*****************************************************************************)
(* Fixed bitwidths extracted from Anthony's models.            *)
(*                                                             *)
(* If you need a size N that is not included here, put the lines  *)
(*                                                             *)
(* type tyN                                                       *)
(* instance (Size tyN) let size = N end                           *)
(* declare isabelle target_rep type tyN = `N`                     *)
```

```
(* declare hol target_rep type tyN = `N`                    *)
(*                                                          *)
(* in your project, replacing N in each line.               *)
(************************************************************)

type TY1
type TY2
type TY3
type TY4
type TY5
type TY6
type TY7
type TY8
type TY9
type TY10
type TY11
type TY12
type TY13
type TY14
type TY15
type TY16
type TY17
type TY18
type TY19
type TY20
type TY21
type TY22
type TY23
type TY24
type TY25
type TY26
type TY27
type TY28
type TY29
type TY30
type TY31
type TY32
type TY33
type TY34
type TY35
type TY36
type TY37
type TY38
type TY39
type TY40
type TY41
type TY42
type TY43
type TY44
type TY45
type TY46
type TY47
type TY48
type TY49
type TY50
type TY51
type TY52
type TY53
```

type TY_{54}
type TY_{55}
type TY_{56}
type TY_{57}
type TY_{58}
type TY_{59}
type TY_{60}
type TY_{61}
type TY_{62}
type TY_{63}
type TY_{64}
type TY_{65}
type TY_{66}
type TY_{67}
type TY_{68}
type TY_{69}
type TY_{70}
type TY_{71}
type TY_{72}
type TY_{73}
type TY_{74}
type TY_{75}
type TY_{76}
type TY_{77}
type TY_{78}
type TY_{79}
type TY_{80}
type TY_{81}
type TY_{82}
type TY_{83}
type TY_{84}
type TY_{85}
type TY_{86}
type TY_{87}
type TY_{88}
type TY_{89}
type TY_{90}
type TY_{91}
type TY_{92}
type TY_{93}
type TY_{94}
type TY_{95}
type TY_{96}
type TY_{97}
type TY_{98}
type TY_{99}
type TY_{100}
type TY_{101}
type TY_{102}
type TY_{103}
type TY_{104}
type TY_{105}
type TY_{106}
type TY_{107}
type TY_{108}
type TY_{109}
type TY_{110}
type TY_{111}

type TY_{112}
type TY_{113}
type TY_{114}
type TY_{115}
type TY_{116}
type TY_{117}
type TY_{118}
type TY_{119}
type TY_{120}
type TY_{121}
type TY_{122}
type TY_{123}
type TY_{124}
type TY_{125}
type TY_{126}
type TY_{127}
type TY_{128}
type TY_{129}
type TY_{130}
type TY_{131}
type TY_{132}
type TY_{133}
type TY_{134}
type TY_{135}
type TY_{136}
type TY_{137}
type TY_{138}
type TY_{139}
type TY_{140}
type TY_{141}
type TY_{142}
type TY_{143}
type TY_{144}
type TY_{145}
type TY_{146}
type TY_{147}
type TY_{148}
type TY_{149}
type TY_{150}
type TY_{151}
type TY_{152}
type TY_{153}
type TY_{154}
type TY_{155}
type TY_{156}
type TY_{157}
type TY_{158}
type TY_{159}
type TY_{160}
type TY_{161}
type TY_{162}
type TY_{163}
type TY_{164}
type TY_{165}
type TY_{166}
type TY_{167}
type TY_{168}
type TY_{169}

type TY_{170}
type TY_{171}
type TY_{172}
type TY_{173}
type TY_{174}
type TY_{175}
type TY_{176}
type TY_{177}
type TY_{178}
type TY_{179}
type TY_{180}
type TY_{181}
type TY_{182}
type TY_{183}
type TY_{184}
type TY_{185}
type TY_{186}
type TY_{187}
type TY_{188}
type TY_{189}
type TY_{190}
type TY_{191}
type TY_{192}
type TY_{193}
type TY_{194}
type TY_{195}
type TY_{196}
type TY_{197}
type TY_{198}
type TY_{199}
type TY_{200}
type TY_{201}
type TY_{202}
type TY_{203}
type TY_{204}
type TY_{205}
type TY_{206}
type TY_{207}
type TY_{208}
type TY_{209}
type TY_{210}
type TY_{211}
type TY_{212}
type TY_{213}
type TY_{214}
type TY_{215}
type TY_{216}
type TY_{217}
type TY_{218}
type TY_{219}
type TY_{220}
type TY_{221}
type TY_{222}
type TY_{223}
type TY_{224}
type TY_{225}
type TY_{226}
type TY_{227}

type TY_{228}
type TY_{229}
type TY_{230}
type TY_{231}
type TY_{232}
type TY_{233}
type TY_{234}
type TY_{235}
type TY_{236}
type TY_{237}
type TY_{238}
type TY_{239}
type TY_{240}
type TY_{241}
type TY_{242}
type TY_{243}
type TY_{244}
type TY_{245}
type TY_{246}
type TY_{247}
type TY_{248}
type TY_{249}
type TY_{250}
type TY_{251}
type TY_{252}
type TY_{253}
type TY_{254}
type TY_{255}
type TY_{256}
type TY_{257}

instance $(Size\ \mathrm{TY}_{1})$ let $size = 1$ end
instance $(Size\ \mathrm{TY}_{2})$ let $size = 2$ end
instance $(Size\ \mathrm{TY}_{3})$ let $size = 3$ end
instance $(Size\ \mathrm{TY}_{4})$ let $size = 4$ end
instance $(Size\ \mathrm{TY}_{5})$ let $size = 5$ end
instance $(Size\ \mathrm{TY}_{6})$ let $size = 6$ end
instance $(Size\ \mathrm{TY}_{7})$ let $size = 7$ end
instance $(Size\ \mathrm{TY}_{8})$ let $size = 8$ end
instance $(Size\ \mathrm{TY}_{9})$ let $size = 9$ end
instance $(Size\ \mathrm{TY}_{10})$ let $size = 10$ end
instance $(Size\ \mathrm{TY}_{11})$ let $size = 11$ end
instance $(Size\ \mathrm{TY}_{12})$ let $size = 12$ end
instance $(Size\ \mathrm{TY}_{13})$ let $size = 13$ end
instance $(Size\ \mathrm{TY}_{14})$ let $size = 14$ end
instance $(Size\ \mathrm{TY}_{15})$ let $size = 15$ end
instance $(Size\ \mathrm{TY}_{16})$ let $size = 16$ end
instance $(Size\ \mathrm{TY}_{17})$ let $size = 17$ end
instance $(Size\ \mathrm{TY}_{18})$ let $size = 18$ end
instance $(Size\ \mathrm{TY}_{19})$ let $size = 19$ end
instance $(Size\ \mathrm{TY}_{20})$ let $size = 20$ end
instance $(Size\ \mathrm{TY}_{21})$ let $size = 21$ end
instance $(Size\ \mathrm{TY}_{22})$ let $size = 22$ end
instance $(Size\ \mathrm{TY}_{23})$ let $size = 23$ end
instance $(Size\ \mathrm{TY}_{24})$ let $size = 24$ end
instance $(Size\ \mathrm{TY}_{25})$ let $size = 25$ end
instance $(Size\ \mathrm{TY}_{26})$ let $size = 26$ end
instance $(Size\ \mathrm{TY}_{27})$ let $size = 27$ end

instance ($Size\ \mathrm{TY}_{28}$) let $size = 28$ end
instance ($Size\ \mathrm{TY}_{29}$) let $size = 29$ end
instance ($Size\ \mathrm{TY}_{30}$) let $size = 30$ end
instance ($Size\ \mathrm{TY}_{31}$) let $size = 31$ end
instance ($Size\ \mathrm{TY}_{32}$) let $size = 32$ end
instance ($Size\ \mathrm{TY}_{33}$) let $size = 33$ end
instance ($Size\ \mathrm{TY}_{34}$) let $size = 34$ end
instance ($Size\ \mathrm{TY}_{35}$) let $size = 35$ end
instance ($Size\ \mathrm{TY}_{36}$) let $size = 36$ end
instance ($Size\ \mathrm{TY}_{37}$) let $size = 37$ end
instance ($Size\ \mathrm{TY}_{38}$) let $size = 38$ end
instance ($Size\ \mathrm{TY}_{39}$) let $size = 39$ end
instance ($Size\ \mathrm{TY}_{40}$) let $size = 40$ end
instance ($Size\ \mathrm{TY}_{41}$) let $size = 41$ end
instance ($Size\ \mathrm{TY}_{42}$) let $size = 42$ end
instance ($Size\ \mathrm{TY}_{43}$) let $size = 43$ end
instance ($Size\ \mathrm{TY}_{44}$) let $size = 44$ end
instance ($Size\ \mathrm{TY}_{45}$) let $size = 45$ end
instance ($Size\ \mathrm{TY}_{46}$) let $size = 46$ end
instance ($Size\ \mathrm{TY}_{47}$) let $size = 47$ end
instance ($Size\ \mathrm{TY}_{48}$) let $size = 48$ end
instance ($Size\ \mathrm{TY}_{49}$) let $size = 49$ end
instance ($Size\ \mathrm{TY}_{50}$) let $size = 50$ end
instance ($Size\ \mathrm{TY}_{51}$) let $size = 51$ end
instance ($Size\ \mathrm{TY}_{52}$) let $size = 52$ end
instance ($Size\ \mathrm{TY}_{53}$) let $size = 53$ end
instance ($Size\ \mathrm{TY}_{54}$) let $size = 54$ end
instance ($Size\ \mathrm{TY}_{55}$) let $size = 55$ end
instance ($Size\ \mathrm{TY}_{56}$) let $size = 56$ end
instance ($Size\ \mathrm{TY}_{57}$) let $size = 57$ end
instance ($Size\ \mathrm{TY}_{58}$) let $size = 58$ end
instance ($Size\ \mathrm{TY}_{59}$) let $size = 59$ end
instance ($Size\ \mathrm{TY}_{60}$) let $size = 60$ end
instance ($Size\ \mathrm{TY}_{61}$) let $size = 61$ end
instance ($Size\ \mathrm{TY}_{62}$) let $size = 62$ end
instance ($Size\ \mathrm{TY}_{63}$) let $size = 63$ end
instance ($Size\ \mathrm{TY}_{64}$) let $size = 64$ end
instance ($Size\ \mathrm{TY}_{65}$) let $size = 65$ end
instance ($Size\ \mathrm{TY}_{66}$) let $size = 66$ end
instance ($Size\ \mathrm{TY}_{67}$) let $size = 67$ end
instance ($Size\ \mathrm{TY}_{68}$) let $size = 68$ end
instance ($Size\ \mathrm{TY}_{69}$) let $size = 69$ end
instance ($Size\ \mathrm{TY}_{70}$) let $size = 70$ end
instance ($Size\ \mathrm{TY}_{71}$) let $size = 71$ end
instance ($Size\ \mathrm{TY}_{72}$) let $size = 72$ end
instance ($Size\ \mathrm{TY}_{73}$) let $size = 73$ end
instance ($Size\ \mathrm{TY}_{74}$) let $size = 74$ end
instance ($Size\ \mathrm{TY}_{75}$) let $size = 75$ end
instance ($Size\ \mathrm{TY}_{76}$) let $size = 76$ end
instance ($Size\ \mathrm{TY}_{77}$) let $size = 77$ end
instance ($Size\ \mathrm{TY}_{78}$) let $size = 78$ end
instance ($Size\ \mathrm{TY}_{79}$) let $size = 79$ end
instance ($Size\ \mathrm{TY}_{80}$) let $size = 80$ end
instance ($Size\ \mathrm{TY}_{81}$) let $size = 81$ end
instance ($Size\ \mathrm{TY}_{82}$) let $size = 82$ end
instance ($Size\ \mathrm{TY}_{83}$) let $size = 83$ end
instance ($Size\ \mathrm{TY}_{84}$) let $size = 84$ end
instance ($Size\ \mathrm{TY}_{85}$) let $size = 85$ end

instance ($Size\ \mathrm{TY}_{86}$) let $size =\ 86$ end
instance ($Size\ \mathrm{TY}_{87}$) let $size =\ 87$ end
instance ($Size\ \mathrm{TY}_{88}$) let $size =\ 88$ end
instance ($Size\ \mathrm{TY}_{89}$) let $size =\ 89$ end
instance ($Size\ \mathrm{TY}_{90}$) let $size =\ 90$ end
instance ($Size\ \mathrm{TY}_{91}$) let $size =\ 91$ end
instance ($Size\ \mathrm{TY}_{92}$) let $size =\ 92$ end
instance ($Size\ \mathrm{TY}_{93}$) let $size =\ 93$ end
instance ($Size\ \mathrm{TY}_{94}$) let $size =\ 94$ end
instance ($Size\ \mathrm{TY}_{95}$) let $size =\ 95$ end
instance ($Size\ \mathrm{TY}_{96}$) let $size =\ 96$ end
instance ($Size\ \mathrm{TY}_{97}$) let $size =\ 97$ end
instance ($Size\ \mathrm{TY}_{98}$) let $size =\ 98$ end
instance ($Size\ \mathrm{TY}_{99}$) let $size =\ 99$ end
instance ($Size\ \mathrm{TY}_{100}$) let $size =\ 100$ end
instance ($Size\ \mathrm{TY}_{101}$) let $size =\ 101$ end
instance ($Size\ \mathrm{TY}_{102}$) let $size =\ 102$ end
instance ($Size\ \mathrm{TY}_{103}$) let $size =\ 103$ end
instance ($Size\ \mathrm{TY}_{104}$) let $size =\ 104$ end
instance ($Size\ \mathrm{TY}_{105}$) let $size =\ 105$ end
instance ($Size\ \mathrm{TY}_{106}$) let $size =\ 106$ end
instance ($Size\ \mathrm{TY}_{107}$) let $size =\ 107$ end
instance ($Size\ \mathrm{TY}_{108}$) let $size =\ 108$ end
instance ($Size\ \mathrm{TY}_{109}$) let $size =\ 109$ end
instance ($Size\ \mathrm{TY}_{110}$) let $size =\ 110$ end
instance ($Size\ \mathrm{TY}_{111}$) let $size =\ 111$ end
instance ($Size\ \mathrm{TY}_{112}$) let $size =\ 112$ end
instance ($Size\ \mathrm{TY}_{113}$) let $size =\ 113$ end
instance ($Size\ \mathrm{TY}_{114}$) let $size =\ 114$ end
instance ($Size\ \mathrm{TY}_{115}$) let $size =\ 115$ end
instance ($Size\ \mathrm{TY}_{116}$) let $size =\ 116$ end
instance ($Size\ \mathrm{TY}_{117}$) let $size =\ 117$ end
instance ($Size\ \mathrm{TY}_{118}$) let $size =\ 118$ end
instance ($Size\ \mathrm{TY}_{119}$) let $size =\ 119$ end
instance ($Size\ \mathrm{TY}_{120}$) let $size =\ 120$ end
instance ($Size\ \mathrm{TY}_{121}$) let $size =\ 121$ end
instance ($Size\ \mathrm{TY}_{122}$) let $size =\ 122$ end
instance ($Size\ \mathrm{TY}_{123}$) let $size =\ 123$ end
instance ($Size\ \mathrm{TY}_{124}$) let $size =\ 124$ end
instance ($Size\ \mathrm{TY}_{125}$) let $size =\ 125$ end
instance ($Size\ \mathrm{TY}_{126}$) let $size =\ 126$ end
instance ($Size\ \mathrm{TY}_{127}$) let $size =\ 127$ end
instance ($Size\ \mathrm{TY}_{128}$) let $size =\ 128$ end
instance ($Size\ \mathrm{TY}_{129}$) let $size =\ 129$ end
instance ($Size\ \mathrm{TY}_{130}$) let $size =\ 130$ end
instance ($Size\ \mathrm{TY}_{131}$) let $size =\ 131$ end
instance ($Size\ \mathrm{TY}_{132}$) let $size =\ 132$ end
instance ($Size\ \mathrm{TY}_{133}$) let $size =\ 133$ end
instance ($Size\ \mathrm{TY}_{134}$) let $size =\ 134$ end
instance ($Size\ \mathrm{TY}_{135}$) let $size =\ 135$ end
instance ($Size\ \mathrm{TY}_{136}$) let $size =\ 136$ end
instance ($Size\ \mathrm{TY}_{137}$) let $size =\ 137$ end
instance ($Size\ \mathrm{TY}_{138}$) let $size =\ 138$ end
instance ($Size\ \mathrm{TY}_{139}$) let $size =\ 139$ end
instance ($Size\ \mathrm{TY}_{140}$) let $size =\ 140$ end
instance ($Size\ \mathrm{TY}_{141}$) let $size =\ 141$ end
instance ($Size\ \mathrm{TY}_{142}$) let $size =\ 142$ end
instance ($Size\ \mathrm{TY}_{143}$) let $size =\ 143$ end

instance ($Size\ \mathrm{TY}_{144}$) let $size =\ 144$ end
instance ($Size\ \mathrm{TY}_{145}$) let $size =\ 145$ end
instance ($Size\ \mathrm{TY}_{146}$) let $size =\ 146$ end
instance ($Size\ \mathrm{TY}_{147}$) let $size =\ 147$ end
instance ($Size\ \mathrm{TY}_{148}$) let $size =\ 148$ end
instance ($Size\ \mathrm{TY}_{149}$) let $size =\ 149$ end
instance ($Size\ \mathrm{TY}_{150}$) let $size =\ 150$ end
instance ($Size\ \mathrm{TY}_{151}$) let $size =\ 151$ end
instance ($Size\ \mathrm{TY}_{152}$) let $size =\ 152$ end
instance ($Size\ \mathrm{TY}_{153}$) let $size =\ 153$ end
instance ($Size\ \mathrm{TY}_{154}$) let $size =\ 154$ end
instance ($Size\ \mathrm{TY}_{155}$) let $size =\ 155$ end
instance ($Size\ \mathrm{TY}_{156}$) let $size =\ 156$ end
instance ($Size\ \mathrm{TY}_{157}$) let $size =\ 157$ end
instance ($Size\ \mathrm{TY}_{158}$) let $size =\ 158$ end
instance ($Size\ \mathrm{TY}_{159}$) let $size =\ 159$ end
instance ($Size\ \mathrm{TY}_{160}$) let $size =\ 160$ end
instance ($Size\ \mathrm{TY}_{161}$) let $size =\ 161$ end
instance ($Size\ \mathrm{TY}_{162}$) let $size =\ 162$ end
instance ($Size\ \mathrm{TY}_{163}$) let $size =\ 163$ end
instance ($Size\ \mathrm{TY}_{164}$) let $size =\ 164$ end
instance ($Size\ \mathrm{TY}_{165}$) let $size =\ 165$ end
instance ($Size\ \mathrm{TY}_{166}$) let $size =\ 166$ end
instance ($Size\ \mathrm{TY}_{167}$) let $size =\ 167$ end
instance ($Size\ \mathrm{TY}_{168}$) let $size =\ 168$ end
instance ($Size\ \mathrm{TY}_{169}$) let $size =\ 169$ end
instance ($Size\ \mathrm{TY}_{170}$) let $size =\ 170$ end
instance ($Size\ \mathrm{TY}_{171}$) let $size =\ 171$ end
instance ($Size\ \mathrm{TY}_{172}$) let $size =\ 172$ end
instance ($Size\ \mathrm{TY}_{173}$) let $size =\ 173$ end
instance ($Size\ \mathrm{TY}_{174}$) let $size =\ 174$ end
instance ($Size\ \mathrm{TY}_{175}$) let $size =\ 175$ end
instance ($Size\ \mathrm{TY}_{176}$) let $size =\ 176$ end
instance ($Size\ \mathrm{TY}_{177}$) let $size =\ 177$ end
instance ($Size\ \mathrm{TY}_{178}$) let $size =\ 178$ end
instance ($Size\ \mathrm{TY}_{179}$) let $size =\ 179$ end
instance ($Size\ \mathrm{TY}_{180}$) let $size =\ 180$ end
instance ($Size\ \mathrm{TY}_{181}$) let $size =\ 181$ end
instance ($Size\ \mathrm{TY}_{182}$) let $size =\ 182$ end
instance ($Size\ \mathrm{TY}_{183}$) let $size =\ 183$ end
instance ($Size\ \mathrm{TY}_{184}$) let $size =\ 184$ end
instance ($Size\ \mathrm{TY}_{185}$) let $size =\ 185$ end
instance ($Size\ \mathrm{TY}_{186}$) let $size =\ 186$ end
instance ($Size\ \mathrm{TY}_{187}$) let $size =\ 187$ end
instance ($Size\ \mathrm{TY}_{188}$) let $size =\ 188$ end
instance ($Size\ \mathrm{TY}_{189}$) let $size =\ 189$ end
instance ($Size\ \mathrm{TY}_{190}$) let $size =\ 190$ end
instance ($Size\ \mathrm{TY}_{191}$) let $size =\ 191$ end
instance ($Size\ \mathrm{TY}_{192}$) let $size =\ 192$ end
instance ($Size\ \mathrm{TY}_{193}$) let $size =\ 193$ end
instance ($Size\ \mathrm{TY}_{194}$) let $size =\ 194$ end
instance ($Size\ \mathrm{TY}_{195}$) let $size =\ 195$ end
instance ($Size\ \mathrm{TY}_{196}$) let $size =\ 196$ end
instance ($Size\ \mathrm{TY}_{197}$) let $size =\ 197$ end
instance ($Size\ \mathrm{TY}_{198}$) let $size =\ 198$ end
instance ($Size\ \mathrm{TY}_{199}$) let $size =\ 199$ end
instance ($Size\ \mathrm{TY}_{200}$) let $size =\ 200$ end
instance ($Size\ \mathrm{TY}_{201}$) let $size =\ 201$ end

instance ($Size\ \mathrm{TY}_{202}$) let $size = \ 202$ end
instance ($Size\ \mathrm{TY}_{203}$) let $size = \ 203$ end
instance ($Size\ \mathrm{TY}_{204}$) let $size = \ 204$ end
instance ($Size\ \mathrm{TY}_{205}$) let $size = \ 205$ end
instance ($Size\ \mathrm{TY}_{206}$) let $size = \ 206$ end
instance ($Size\ \mathrm{TY}_{207}$) let $size = \ 207$ end
instance ($Size\ \mathrm{TY}_{208}$) let $size = \ 208$ end
instance ($Size\ \mathrm{TY}_{209}$) let $size = \ 209$ end
instance ($Size\ \mathrm{TY}_{210}$) let $size = \ 210$ end
instance ($Size\ \mathrm{TY}_{211}$) let $size = \ 211$ end
instance ($Size\ \mathrm{TY}_{212}$) let $size = \ 212$ end
instance ($Size\ \mathrm{TY}_{213}$) let $size = \ 213$ end
instance ($Size\ \mathrm{TY}_{214}$) let $size = \ 214$ end
instance ($Size\ \mathrm{TY}_{215}$) let $size = \ 215$ end
instance ($Size\ \mathrm{TY}_{216}$) let $size = \ 216$ end
instance ($Size\ \mathrm{TY}_{217}$) let $size = \ 217$ end
instance ($Size\ \mathrm{TY}_{218}$) let $size = \ 218$ end
instance ($Size\ \mathrm{TY}_{219}$) let $size = \ 219$ end
instance ($Size\ \mathrm{TY}_{220}$) let $size = \ 220$ end
instance ($Size\ \mathrm{TY}_{221}$) let $size = \ 221$ end
instance ($Size\ \mathrm{TY}_{222}$) let $size = \ 222$ end
instance ($Size\ \mathrm{TY}_{223}$) let $size = \ 223$ end
instance ($Size\ \mathrm{TY}_{224}$) let $size = \ 224$ end
instance ($Size\ \mathrm{TY}_{225}$) let $size = \ 225$ end
instance ($Size\ \mathrm{TY}_{226}$) let $size = \ 226$ end
instance ($Size\ \mathrm{TY}_{227}$) let $size = \ 227$ end
instance ($Size\ \mathrm{TY}_{228}$) let $size = \ 228$ end
instance ($Size\ \mathrm{TY}_{229}$) let $size = \ 229$ end
instance ($Size\ \mathrm{TY}_{230}$) let $size = \ 230$ end
instance ($Size\ \mathrm{TY}_{231}$) let $size = \ 231$ end
instance ($Size\ \mathrm{TY}_{232}$) let $size = \ 232$ end
instance ($Size\ \mathrm{TY}_{233}$) let $size = \ 233$ end
instance ($Size\ \mathrm{TY}_{234}$) let $size = \ 234$ end
instance ($Size\ \mathrm{TY}_{235}$) let $size = \ 235$ end
instance ($Size\ \mathrm{TY}_{236}$) let $size = \ 236$ end
instance ($Size\ \mathrm{TY}_{237}$) let $size = \ 237$ end
instance ($Size\ \mathrm{TY}_{238}$) let $size = \ 238$ end
instance ($Size\ \mathrm{TY}_{239}$) let $size = \ 239$ end
instance ($Size\ \mathrm{TY}_{240}$) let $size = \ 240$ end
instance ($Size\ \mathrm{TY}_{241}$) let $size = \ 241$ end
instance ($Size\ \mathrm{TY}_{242}$) let $size = \ 242$ end
instance ($Size\ \mathrm{TY}_{243}$) let $size = \ 243$ end
instance ($Size\ \mathrm{TY}_{244}$) let $size = \ 244$ end
instance ($Size\ \mathrm{TY}_{245}$) let $size = \ 245$ end
instance ($Size\ \mathrm{TY}_{246}$) let $size = \ 246$ end
instance ($Size\ \mathrm{TY}_{247}$) let $size = \ 247$ end
instance ($Size\ \mathrm{TY}_{248}$) let $size = \ 248$ end
instance ($Size\ \mathrm{TY}_{249}$) let $size = \ 249$ end
instance ($Size\ \mathrm{TY}_{250}$) let $size = \ 250$ end
instance ($Size\ \mathrm{TY}_{251}$) let $size = \ 251$ end
instance ($Size\ \mathrm{TY}_{252}$) let $size = \ 252$ end
instance ($Size\ \mathrm{TY}_{253}$) let $size = \ 253$ end
instance ($Size\ \mathrm{TY}_{254}$) let $size = \ 254$ end
instance ($Size\ \mathrm{TY}_{255}$) let $size = \ 255$ end
instance ($Size\ \mathrm{TY}_{256}$) let $size = \ 256$ end
instance ($Size\ \mathrm{TY}_{257}$) let $size = \ 257$ end

declare $isabelle$ target_rep type TY_{1} = '1'

declare *isabelle* target_rep type TY_{2} = '2'
declare *isabelle* target_rep type TY_{3} = '3'
declare *isabelle* target_rep type TY_{4} = '4'
declare *isabelle* target_rep type TY_{5} = '5'
declare *isabelle* target_rep type TY_{6} = '6'
declare *isabelle* target_rep type TY_{7} = '7'
declare *isabelle* target_rep type TY_{8} = '8'
declare *isabelle* target_rep type TY_{9} = '9'
declare *isabelle* target_rep type TY_{10} = '10'
declare *isabelle* target_rep type TY_{11} = '11'
declare *isabelle* target_rep type TY_{12} = '12'
declare *isabelle* target_rep type TY_{13} = '13'
declare *isabelle* target_rep type TY_{14} = '14'
declare *isabelle* target_rep type TY_{15} = '15'
declare *isabelle* target_rep type TY_{16} = '16'
declare *isabelle* target_rep type TY_{17} = '17'
declare *isabelle* target_rep type TY_{18} = '18'
declare *isabelle* target_rep type TY_{19} = '19'
declare *isabelle* target_rep type TY_{20} = '20'
declare *isabelle* target_rep type TY_{21} = '21'
declare *isabelle* target_rep type TY_{22} = '22'
declare *isabelle* target_rep type TY_{23} = '23'
declare *isabelle* target_rep type TY_{24} = '24'
declare *isabelle* target_rep type TY_{25} = '25'
declare *isabelle* target_rep type TY_{26} = '26'
declare *isabelle* target_rep type TY_{27} = '27'
declare *isabelle* target_rep type TY_{28} = '28'
declare *isabelle* target_rep type TY_{29} = '29'
declare *isabelle* target_rep type TY_{30} = '30'
declare *isabelle* target_rep type TY_{31} = '31'
declare *isabelle* target_rep type TY_{32} = '32'
declare *isabelle* target_rep type TY_{33} = '33'
declare *isabelle* target_rep type TY_{34} = '34'
declare *isabelle* target_rep type TY_{35} = '35'
declare *isabelle* target_rep type TY_{36} = '36'
declare *isabelle* target_rep type TY_{37} = '37'
declare *isabelle* target_rep type TY_{38} = '38'
declare *isabelle* target_rep type TY_{39} = '39'
declare *isabelle* target_rep type TY_{40} = '40'
declare *isabelle* target_rep type TY_{41} = '41'
declare *isabelle* target_rep type TY_{42} = '42'
declare *isabelle* target_rep type TY_{43} = '43'
declare *isabelle* target_rep type TY_{44} = '44'
declare *isabelle* target_rep type TY_{45} = '45'
declare *isabelle* target_rep type TY_{46} = '46'
declare *isabelle* target_rep type TY_{47} = '47'
declare *isabelle* target_rep type TY_{48} = '48'
declare *isabelle* target_rep type TY_{49} = '49'
declare *isabelle* target_rep type TY_{50} = '50'
declare *isabelle* target_rep type TY_{51} = '51'
declare *isabelle* target_rep type TY_{52} = '52'
declare *isabelle* target_rep type TY_{53} = '53'
declare *isabelle* target_rep type TY_{54} = '54'
declare *isabelle* target_rep type TY_{55} = '55'
declare *isabelle* target_rep type TY_{56} = '56'
declare *isabelle* target_rep type TY_{57} = '57'
declare *isabelle* target_rep type TY_{58} = '58'
declare *isabelle* target_rep type TY_{59} = '59'

declare *isabelle* target_rep type TY_{60} = '60'
declare *isabelle* target_rep type TY_{61} = '61'
declare *isabelle* target_rep type TY_{62} = '62'
declare *isabelle* target_rep type TY_{63} = '63'
declare *isabelle* target_rep type TY_{64} = '64'
declare *isabelle* target_rep type TY_{65} = '65'
declare *isabelle* target_rep type TY_{66} = '66'
declare *isabelle* target_rep type TY_{67} = '67'
declare *isabelle* target_rep type TY_{68} = '68'
declare *isabelle* target_rep type TY_{69} = '69'
declare *isabelle* target_rep type TY_{70} = '70'
declare *isabelle* target_rep type TY_{71} = '71'
declare *isabelle* target_rep type TY_{72} = '72'
declare *isabelle* target_rep type TY_{73} = '73'
declare *isabelle* target_rep type TY_{74} = '74'
declare *isabelle* target_rep type TY_{75} = '75'
declare *isabelle* target_rep type TY_{76} = '76'
declare *isabelle* target_rep type TY_{77} = '77'
declare *isabelle* target_rep type TY_{78} = '78'
declare *isabelle* target_rep type TY_{79} = '79'
declare *isabelle* target_rep type TY_{80} = '80'
declare *isabelle* target_rep type TY_{81} = '81'
declare *isabelle* target_rep type TY_{82} = '82'
declare *isabelle* target_rep type TY_{83} = '83'
declare *isabelle* target_rep type TY_{84} = '84'
declare *isabelle* target_rep type TY_{85} = '85'
declare *isabelle* target_rep type TY_{86} = '86'
declare *isabelle* target_rep type TY_{87} = '87'
declare *isabelle* target_rep type TY_{88} = '88'
declare *isabelle* target_rep type TY_{89} = '89'
declare *isabelle* target_rep type TY_{90} = '90'
declare *isabelle* target_rep type TY_{91} = '91'
declare *isabelle* target_rep type TY_{92} = '92'
declare *isabelle* target_rep type TY_{93} = '93'
declare *isabelle* target_rep type TY_{94} = '94'
declare *isabelle* target_rep type TY_{95} = '95'
declare *isabelle* target_rep type TY_{96} = '96'
declare *isabelle* target_rep type TY_{97} = '97'
declare *isabelle* target_rep type TY_{98} = '98'
declare *isabelle* target_rep type TY_{99} = '99'
declare *isabelle* target_rep type TY_{100} = '100'
declare *isabelle* target_rep type TY_{101} = '101'
declare *isabelle* target_rep type TY_{102} = '102'
declare *isabelle* target_rep type TY_{103} = '103'
declare *isabelle* target_rep type TY_{104} = '104'
declare *isabelle* target_rep type TY_{105} = '105'
declare *isabelle* target_rep type TY_{106} = '106'
declare *isabelle* target_rep type TY_{107} = '107'
declare *isabelle* target_rep type TY_{108} = '108'
declare *isabelle* target_rep type TY_{109} = '109'
declare *isabelle* target_rep type TY_{110} = '110'
declare *isabelle* target_rep type TY_{111} = '111'
declare *isabelle* target_rep type TY_{112} = '112'
declare *isabelle* target_rep type TY_{113} = '113'
declare *isabelle* target_rep type TY_{114} = '114'
declare *isabelle* target_rep type TY_{115} = '115'
declare *isabelle* target_rep type TY_{116} = '116'
declare *isabelle* target_rep type TY_{117} = '117'

declare *isabelle* target_rep type TY_{118} = '118'
declare *isabelle* target_rep type TY_{119} = '119'
declare *isabelle* target_rep type TY_{120} = '120'
declare *isabelle* target_rep type TY_{121} = '121'
declare *isabelle* target_rep type TY_{122} = '122'
declare *isabelle* target_rep type TY_{123} = '123'
declare *isabelle* target_rep type TY_{124} = '124'
declare *isabelle* target_rep type TY_{125} = '125'
declare *isabelle* target_rep type TY_{126} = '126'
declare *isabelle* target_rep type TY_{127} = '127'
declare *isabelle* target_rep type TY_{128} = '128'
declare *isabelle* target_rep type TY_{129} = '129'
declare *isabelle* target_rep type TY_{130} = '130'
declare *isabelle* target_rep type TY_{131} = '131'
declare *isabelle* target_rep type TY_{132} = '132'
declare *isabelle* target_rep type TY_{133} = '133'
declare *isabelle* target_rep type TY_{134} = '134'
declare *isabelle* target_rep type TY_{135} = '135'
declare *isabelle* target_rep type TY_{136} = '136'
declare *isabelle* target_rep type TY_{137} = '137'
declare *isabelle* target_rep type TY_{138} = '138'
declare *isabelle* target_rep type TY_{139} = '139'
declare *isabelle* target_rep type TY_{140} = '140'
declare *isabelle* target_rep type TY_{141} = '141'
declare *isabelle* target_rep type TY_{142} = '142'
declare *isabelle* target_rep type TY_{143} = '143'
declare *isabelle* target_rep type TY_{144} = '144'
declare *isabelle* target_rep type TY_{145} = '145'
declare *isabelle* target_rep type TY_{146} = '146'
declare *isabelle* target_rep type TY_{147} = '147'
declare *isabelle* target_rep type TY_{148} = '148'
declare *isabelle* target_rep type TY_{149} = '149'
declare *isabelle* target_rep type TY_{150} = '150'
declare *isabelle* target_rep type TY_{151} = '151'
declare *isabelle* target_rep type TY_{152} = '152'
declare *isabelle* target_rep type TY_{153} = '153'
declare *isabelle* target_rep type TY_{154} = '154'
declare *isabelle* target_rep type TY_{155} = '155'
declare *isabelle* target_rep type TY_{156} = '156'
declare *isabelle* target_rep type TY_{157} = '157'
declare *isabelle* target_rep type TY_{158} = '158'
declare *isabelle* target_rep type TY_{159} = '159'
declare *isabelle* target_rep type TY_{160} = '160'
declare *isabelle* target_rep type TY_{161} = '161'
declare *isabelle* target_rep type TY_{162} = '162'
declare *isabelle* target_rep type TY_{163} = '163'
declare *isabelle* target_rep type TY_{164} = '164'
declare *isabelle* target_rep type TY_{165} = '165'
declare *isabelle* target_rep type TY_{166} = '166'
declare *isabelle* target_rep type TY_{167} = '167'
declare *isabelle* target_rep type TY_{168} = '168'
declare *isabelle* target_rep type TY_{169} = '169'
declare *isabelle* target_rep type TY_{170} = '170'
declare *isabelle* target_rep type TY_{171} = '171'
declare *isabelle* target_rep type TY_{172} = '172'
declare *isabelle* target_rep type TY_{173} = '173'
declare *isabelle* target_rep type TY_{174} = '174'
declare *isabelle* target_rep type TY_{175} = '175'

declare *isabelle* target_rep type TY_{176} = '176'
declare *isabelle* target_rep type TY_{177} = '177'
declare *isabelle* target_rep type TY_{178} = '178'
declare *isabelle* target_rep type TY_{179} = '179'
declare *isabelle* target_rep type TY_{180} = '180'
declare *isabelle* target_rep type TY_{181} = '181'
declare *isabelle* target_rep type TY_{182} = '182'
declare *isabelle* target_rep type TY_{183} = '183'
declare *isabelle* target_rep type TY_{184} = '184'
declare *isabelle* target_rep type TY_{185} = '185'
declare *isabelle* target_rep type TY_{186} = '186'
declare *isabelle* target_rep type TY_{187} = '187'
declare *isabelle* target_rep type TY_{188} = '188'
declare *isabelle* target_rep type TY_{189} = '189'
declare *isabelle* target_rep type TY_{190} = '190'
declare *isabelle* target_rep type TY_{191} = '191'
declare *isabelle* target_rep type TY_{192} = '192'
declare *isabelle* target_rep type TY_{193} = '193'
declare *isabelle* target_rep type TY_{194} = '194'
declare *isabelle* target_rep type TY_{195} = '195'
declare *isabelle* target_rep type TY_{196} = '196'
declare *isabelle* target_rep type TY_{197} = '197'
declare *isabelle* target_rep type TY_{198} = '198'
declare *isabelle* target_rep type TY_{199} = '199'
declare *isabelle* target_rep type TY_{200} = '200'
declare *isabelle* target_rep type TY_{201} = '201'
declare *isabelle* target_rep type TY_{202} = '202'
declare *isabelle* target_rep type TY_{203} = '203'
declare *isabelle* target_rep type TY_{204} = '204'
declare *isabelle* target_rep type TY_{205} = '205'
declare *isabelle* target_rep type TY_{206} = '206'
declare *isabelle* target_rep type TY_{207} = '207'
declare *isabelle* target_rep type TY_{208} = '208'
declare *isabelle* target_rep type TY_{209} = '209'
declare *isabelle* target_rep type TY_{210} = '210'
declare *isabelle* target_rep type TY_{211} = '211'
declare *isabelle* target_rep type TY_{212} = '212'
declare *isabelle* target_rep type TY_{213} = '213'
declare *isabelle* target_rep type TY_{214} = '214'
declare *isabelle* target_rep type TY_{215} = '215'
declare *isabelle* target_rep type TY_{216} = '216'
declare *isabelle* target_rep type TY_{217} = '217'
declare *isabelle* target_rep type TY_{218} = '218'
declare *isabelle* target_rep type TY_{219} = '219'
declare *isabelle* target_rep type TY_{220} = '220'
declare *isabelle* target_rep type TY_{221} = '221'
declare *isabelle* target_rep type TY_{222} = '222'
declare *isabelle* target_rep type TY_{223} = '223'
declare *isabelle* target_rep type TY_{224} = '224'
declare *isabelle* target_rep type TY_{225} = '225'
declare *isabelle* target_rep type TY_{226} = '226'
declare *isabelle* target_rep type TY_{227} = '227'
declare *isabelle* target_rep type TY_{228} = '228'
declare *isabelle* target_rep type TY_{229} = '229'
declare *isabelle* target_rep type TY_{230} = '230'
declare *isabelle* target_rep type TY_{231} = '231'
declare *isabelle* target_rep type TY_{232} = '232'
declare *isabelle* target_rep type TY_{233} = '233'

declare *isabelle* target_rep type TY_{234} = '234'
declare *isabelle* target_rep type TY_{235} = '235'
declare *isabelle* target_rep type TY_{236} = '236'
declare *isabelle* target_rep type TY_{237} = '237'
declare *isabelle* target_rep type TY_{238} = '238'
declare *isabelle* target_rep type TY_{239} = '239'
declare *isabelle* target_rep type TY_{240} = '240'
declare *isabelle* target_rep type TY_{241} = '241'
declare *isabelle* target_rep type TY_{242} = '242'
declare *isabelle* target_rep type TY_{243} = '243'
declare *isabelle* target_rep type TY_{244} = '244'
declare *isabelle* target_rep type TY_{245} = '245'
declare *isabelle* target_rep type TY_{246} = '246'
declare *isabelle* target_rep type TY_{247} = '247'
declare *isabelle* target_rep type TY_{248} = '248'
declare *isabelle* target_rep type TY_{249} = '249'
declare *isabelle* target_rep type TY_{250} = '250'
declare *isabelle* target_rep type TY_{251} = '251'
declare *isabelle* target_rep type TY_{252} = '252'
declare *isabelle* target_rep type TY_{253} = '253'
declare *isabelle* target_rep type TY_{254} = '254'
declare *isabelle* target_rep type TY_{255} = '255'
declare *isabelle* target_rep type TY_{256} = '256'
declare *isabelle* target_rep type TY_{257} = '257'

declare *hol* target_rep type TY_{1} = '1'
declare *hol* target_rep type TY_{2} = '2'
declare *hol* target_rep type TY_{3} = '3'
declare *hol* target_rep type TY_{4} = '4'
declare *hol* target_rep type TY_{5} = '5'
declare *hol* target_rep type TY_{6} = '6'
declare *hol* target_rep type TY_{7} = '7'
declare *hol* target_rep type TY_{8} = '8'
declare *hol* target_rep type TY_{9} = '9'
declare *hol* target_rep type TY_{10} = '10'
declare *hol* target_rep type TY_{11} = '11'
declare *hol* target_rep type TY_{12} = '12'
declare *hol* target_rep type TY_{13} = '13'
declare *hol* target_rep type TY_{14} = '14'
declare *hol* target_rep type TY_{15} = '15'
declare *hol* target_rep type TY_{16} = '16'
declare *hol* target_rep type TY_{17} = '17'
declare *hol* target_rep type TY_{18} = '18'
declare *hol* target_rep type TY_{19} = '19'
declare *hol* target_rep type TY_{20} = '20'
declare *hol* target_rep type TY_{21} = '21'
declare *hol* target_rep type TY_{22} = '22'
declare *hol* target_rep type TY_{23} = '23'
declare *hol* target_rep type TY_{24} = '24'
declare *hol* target_rep type TY_{25} = '25'
declare *hol* target_rep type TY_{26} = '26'
declare *hol* target_rep type TY_{27} = '27'
declare *hol* target_rep type TY_{28} = '28'
declare *hol* target_rep type TY_{29} = '29'
declare *hol* target_rep type TY_{30} = '30'
declare *hol* target_rep type TY_{31} = '31'
declare *hol* target_rep type TY_{32} = '32'
declare *hol* target_rep type TY_{33} = '33'

declare hol target_rep type TY_{34} = '34'
declare hol target_rep type TY_{35} = '35'
declare hol target_rep type TY_{36} = '36'
declare hol target_rep type TY_{37} = '37'
declare hol target_rep type TY_{38} = '38'
declare hol target_rep type TY_{39} = '39'
declare hol target_rep type TY_{40} = '40'
declare hol target_rep type TY_{41} = '41'
declare hol target_rep type TY_{42} = '42'
declare hol target_rep type TY_{43} = '43'
declare hol target_rep type TY_{44} = '44'
declare hol target_rep type TY_{45} = '45'
declare hol target_rep type TY_{46} = '46'
declare hol target_rep type TY_{47} = '47'
declare hol target_rep type TY_{48} = '48'
declare hol target_rep type TY_{49} = '49'
declare hol target_rep type TY_{50} = '50'
declare hol target_rep type TY_{51} = '51'
declare hol target_rep type TY_{52} = '52'
declare hol target_rep type TY_{53} = '53'
declare hol target_rep type TY_{54} = '54'
declare hol target_rep type TY_{55} = '55'
declare hol target_rep type TY_{56} = '56'
declare hol target_rep type TY_{57} = '57'
declare hol target_rep type TY_{58} = '58'
declare hol target_rep type TY_{59} = '59'
declare hol target_rep type TY_{60} = '60'
declare hol target_rep type TY_{61} = '61'
declare hol target_rep type TY_{62} = '62'
declare hol target_rep type TY_{63} = '63'
declare hol target_rep type TY_{64} = '64'
declare hol target_rep type TY_{65} = '65'
declare hol target_rep type TY_{66} = '66'
declare hol target_rep type TY_{67} = '67'
declare hol target_rep type TY_{68} = '68'
declare hol target_rep type TY_{69} = '69'
declare hol target_rep type TY_{70} = '70'
declare hol target_rep type TY_{71} = '71'
declare hol target_rep type TY_{72} = '72'
declare hol target_rep type TY_{73} = '73'
declare hol target_rep type TY_{74} = '74'
declare hol target_rep type TY_{75} = '75'
declare hol target_rep type TY_{76} = '76'
declare hol target_rep type TY_{77} = '77'
declare hol target_rep type TY_{78} = '78'
declare hol target_rep type TY_{79} = '79'
declare hol target_rep type TY_{80} = '80'
declare hol target_rep type TY_{81} = '81'
declare hol target_rep type TY_{82} = '82'
declare hol target_rep type TY_{83} = '83'
declare hol target_rep type TY_{84} = '84'
declare hol target_rep type TY_{85} = '85'
declare hol target_rep type TY_{86} = '86'
declare hol target_rep type TY_{87} = '87'
declare hol target_rep type TY_{88} = '88'
declare hol target_rep type TY_{89} = '89'
declare hol target_rep type TY_{90} = '90'
declare hol target_rep type TY_{91} = '91'

declare hol target_rep type TY_{92} = '92'
declare hol target_rep type TY_{93} = '93'
declare hol target_rep type TY_{94} = '94'
declare hol target_rep type TY_{95} = '95'
declare hol target_rep type TY_{96} = '96'
declare hol target_rep type TY_{97} = '97'
declare hol target_rep type TY_{98} = '98'
declare hol target_rep type TY_{99} = '99'
declare hol target_rep type TY_{100} = '100'
declare hol target_rep type TY_{101} = '101'
declare hol target_rep type TY_{102} = '102'
declare hol target_rep type TY_{103} = '103'
declare hol target_rep type TY_{104} = '104'
declare hol target_rep type TY_{105} = '105'
declare hol target_rep type TY_{106} = '106'
declare hol target_rep type TY_{107} = '107'
declare hol target_rep type TY_{108} = '108'
declare hol target_rep type TY_{109} = '109'
declare hol target_rep type TY_{110} = '110'
declare hol target_rep type TY_{111} = '111'
declare hol target_rep type TY_{112} = '112'
declare hol target_rep type TY_{113} = '113'
declare hol target_rep type TY_{114} = '114'
declare hol target_rep type TY_{115} = '115'
declare hol target_rep type TY_{116} = '116'
declare hol target_rep type TY_{117} = '117'
declare hol target_rep type TY_{118} = '118'
declare hol target_rep type TY_{119} = '119'
declare hol target_rep type TY_{120} = '120'
declare hol target_rep type TY_{121} = '121'
declare hol target_rep type TY_{122} = '122'
declare hol target_rep type TY_{123} = '123'
declare hol target_rep type TY_{124} = '124'
declare hol target_rep type TY_{125} = '125'
declare hol target_rep type TY_{126} = '126'
declare hol target_rep type TY_{127} = '127'
declare hol target_rep type TY_{128} = '128'
declare hol target_rep type TY_{129} = '129'
declare hol target_rep type TY_{130} = '130'
declare hol target_rep type TY_{131} = '131'
declare hol target_rep type TY_{132} = '132'
declare hol target_rep type TY_{133} = '133'
declare hol target_rep type TY_{134} = '134'
declare hol target_rep type TY_{135} = '135'
declare hol target_rep type TY_{136} = '136'
declare hol target_rep type TY_{137} = '137'
declare hol target_rep type TY_{138} = '138'
declare hol target_rep type TY_{139} = '139'
declare hol target_rep type TY_{140} = '140'
declare hol target_rep type TY_{141} = '141'
declare hol target_rep type TY_{142} = '142'
declare hol target_rep type TY_{143} = '143'
declare hol target_rep type TY_{144} = '144'
declare hol target_rep type TY_{145} = '145'
declare hol target_rep type TY_{146} = '146'
declare hol target_rep type TY_{147} = '147'
declare hol target_rep type TY_{148} = '148'
declare hol target_rep type TY_{149} = '149'

declare hol target_rep type TY_{150} = '150'
declare hol target_rep type TY_{151} = '151'
declare hol target_rep type TY_{152} = '152'
declare hol target_rep type TY_{153} = '153'
declare hol target_rep type TY_{154} = '154'
declare hol target_rep type TY_{155} = '155'
declare hol target_rep type TY_{156} = '156'
declare hol target_rep type TY_{157} = '157'
declare hol target_rep type TY_{158} = '158'
declare hol target_rep type TY_{159} = '159'
declare hol target_rep type TY_{160} = '160'
declare hol target_rep type TY_{161} = '161'
declare hol target_rep type TY_{162} = '162'
declare hol target_rep type TY_{163} = '163'
declare hol target_rep type TY_{164} = '164'
declare hol target_rep type TY_{165} = '165'
declare hol target_rep type TY_{166} = '166'
declare hol target_rep type TY_{167} = '167'
declare hol target_rep type TY_{168} = '168'
declare hol target_rep type TY_{169} = '169'
declare hol target_rep type TY_{170} = '170'
declare hol target_rep type TY_{171} = '171'
declare hol target_rep type TY_{172} = '172'
declare hol target_rep type TY_{173} = '173'
declare hol target_rep type TY_{174} = '174'
declare hol target_rep type TY_{175} = '175'
declare hol target_rep type TY_{176} = '176'
declare hol target_rep type TY_{177} = '177'
declare hol target_rep type TY_{178} = '178'
declare hol target_rep type TY_{179} = '179'
declare hol target_rep type TY_{180} = '180'
declare hol target_rep type TY_{181} = '181'
declare hol target_rep type TY_{182} = '182'
declare hol target_rep type TY_{183} = '183'
declare hol target_rep type TY_{184} = '184'
declare hol target_rep type TY_{185} = '185'
declare hol target_rep type TY_{186} = '186'
declare hol target_rep type TY_{187} = '187'
declare hol target_rep type TY_{188} = '188'
declare hol target_rep type TY_{189} = '189'
declare hol target_rep type TY_{190} = '190'
declare hol target_rep type TY_{191} = '191'
declare hol target_rep type TY_{192} = '192'
declare hol target_rep type TY_{193} = '193'
declare hol target_rep type TY_{194} = '194'
declare hol target_rep type TY_{195} = '195'
declare hol target_rep type TY_{196} = '196'
declare hol target_rep type TY_{197} = '197'
declare hol target_rep type TY_{198} = '198'
declare hol target_rep type TY_{199} = '199'
declare hol target_rep type TY_{200} = '200'
declare hol target_rep type TY_{201} = '201'
declare hol target_rep type TY_{202} = '202'
declare hol target_rep type TY_{203} = '203'
declare hol target_rep type TY_{204} = '204'
declare hol target_rep type TY_{205} = '205'
declare hol target_rep type TY_{206} = '206'
declare hol target_rep type TY_{207} = '207'

```
declare hol target_rep type TY208 = '208'
declare hol target_rep type TY209 = '209'
declare hol target_rep type TY210 = '210'
declare hol target_rep type TY211 = '211'
declare hol target_rep type TY212 = '212'
declare hol target_rep type TY213 = '213'
declare hol target_rep type TY214 = '214'
declare hol target_rep type TY215 = '215'
declare hol target_rep type TY216 = '216'
declare hol target_rep type TY217 = '217'
declare hol target_rep type TY218 = '218'
declare hol target_rep type TY219 = '219'
declare hol target_rep type TY220 = '220'
declare hol target_rep type TY221 = '221'
declare hol target_rep type TY222 = '222'
declare hol target_rep type TY223 = '223'
declare hol target_rep type TY224 = '224'
declare hol target_rep type TY225 = '225'
declare hol target_rep type TY226 = '226'
declare hol target_rep type TY227 = '227'
declare hol target_rep type TY228 = '228'
declare hol target_rep type TY229 = '229'
declare hol target_rep type TY230 = '230'
declare hol target_rep type TY231 = '231'
declare hol target_rep type TY232 = '232'
declare hol target_rep type TY233 = '233'
declare hol target_rep type TY234 = '234'
declare hol target_rep type TY235 = '235'
declare hol target_rep type TY236 = '236'
declare hol target_rep type TY237 = '237'
declare hol target_rep type TY238 = '238'
declare hol target_rep type TY239 = '239'
declare hol target_rep type TY240 = '240'
declare hol target_rep type TY241 = '241'
declare hol target_rep type TY242 = '242'
declare hol target_rep type TY243 = '243'
declare hol target_rep type TY244 = '244'
declare hol target_rep type TY245 = '245'
declare hol target_rep type TY246 = '246'
declare hol target_rep type TY247 = '247'
declare hol target_rep type TY248 = '248'
declare hol target_rep type TY249 = '249'
declare hol target_rep type TY250 = '250'
declare hol target_rep type TY251 = '251'
declare hol target_rep type TY252 = '252'
declare hol target_rep type TY253 = '253'
declare hol target_rep type TY254 = '254'
declare hol target_rep type TY255 = '255'
declare hol target_rep type TY256 = '256'
declare hol target_rep type TY257 = '257'

val word_length : ∀ α. MWORD α → NAT
declare ocaml target_rep function word_length = 'Lem.word_length'
declare isabelle target_rep function word_length = 'size'
declare hol target_rep function word_length = 'words$word_len'

(* ********************************************************************** *)
(* Conversions                                                            *)
```

```
(* ********************************************************************** *)

val signedIntegerFromWord : ∀ α. MWORD α → $\MATHBB{Z}$

declare isabelle target_rep function signedIntegerFromWord = 'Word.sint'
declare hol target_rep function signedIntegerFromWord = 'integer_word$w2i'
declare ocaml target_rep function signedIntegerFromWord = 'Lem.signedIntegerFromWord'

val unsignedIntegerFromWord : ∀ α. MWORD α → $\MATHBB{Z}$

declare isabelle target_rep function unsignedIntegerFromWord = 'Word.uint'
declare hol target_rep function unsignedIntegerFromWord = 'lem$w2ui'
declare ocaml target_rep function unsignedIntegerFromWord = 'Lem.naturalFromWord'

(* Version without typeclass constraint so that we can derive operations  in Lem for one of the theorem prove
val proverWordFromInteger : ∀ α. $\MATHBB{Z}$ → MWORD α

declare isabelle target_rep function proverWordFromInteger = 'Word.word_of_int'
declare hol target_rep function proverWordFromInteger = 'integer_word$i2w'
declare coq target_rep function proverWordFromInteger = 'DAEMON'

val wordFromInteger : ∀ α. Size α ⇒ $\MATHBB{Z}$ → MWORD α

let inline {isabelle, hol, coq} wordFromInteger i = proverWordFromInteger i
(* The OCaml version is defined after the arithmetic operations, below. *)

val naturalFromWord : ∀ α. MWORD α → $\MATHBB{N}$

declare isabelle target_rep function naturalFromWord = 'Word.unat'
declare hol target_rep function naturalFromWord = 'words$w2n'
declare ocaml target_rep function naturalFromWord = 'Lem.naturalFromWord'

val wordFromNatural : ∀ α. Size α ⇒ $\MATHBB{N}$ → MWORD α

declare hol target_rep function wordFromNatural = 'words$n2w'

let inline {isabelle} wordFromNatural n =
  wordFromInteger (integerFromNatural n)

let {ocaml} wordFromNatural n =  ocaml_inject (size,  n)

val wordToHex : ∀ α. MWORD α → STRING
declare hol target_rep function wordToHex = 'words$word_to_hex_string'
(* Building libraries fails if we don't provide implementations for the  type class. *)
let {ocaml, isabelle, coq} wordToHex w =  "wordToHex not yet implemented"

instance ∀ α. (Show (MWORD α))
  let show =  wordToHex
end

val wordFromBitlist : ∀ α. Size α ⇒ LIST BOOL → MWORD α
declare isabelle target_rep function wordFromBitlist = 'Word.of_bl'
declare hol target_rep function wordFromBitlist = 'bitstring$v2w'
declare ocaml target_rep function wordFromBitlist = 'Lem.wordFromBitlist'

val bitlistFromWord : ∀ α. MWORD α → LIST BOOL
declare isabelle target_rep function bitlistFromWord = 'Word.to_bl'
declare hol target_rep function bitlistFromWord = 'bitstring$w2v'
```

```
declare ocaml target_rep function bitlistFromWord = 'Lem.bitlistFromWord'


val size_test_fn : ∀ α. Size α ⇒ MWORD α → NAT
let size_test_fn _ = size

assert {ocaml, isabelle} size_test : size_test_fn ((wordFromNatural 0) : MWORD TY5) = 5

assert {ocaml, isabelle, hol} size_itself_test : size_itself (the_value : ITSELF TY7) = 7

assert {ocaml, hol, isabelle} length_test :
  word_length ((wordFromNatural 0) : MWORD TY13) = 13
assert {ocaml, hol, isabelle} signedIntFromword_test :
  signedIntegerFromWord ((wordFromNatural 130) : MWORD TY8) = −126
assert {ocaml, hol, isabelle} wordFromBitlist_test :
  ((wordFromBitlist [false; false; true; false]) : MWORD TY4) = wordFromNatural 2
assert {ocaml, hol, isabelle} bitlistFromWord_test :
  bitlistFromWord ((wordFromNatural 2) : MWORD TY4) = [false; false; true; false]
assert {ocaml, hol, isabelle} wordFromBitlist_bitListFromWord_test :
  let w : MWORD TY8 = wordFromNatural 33 in
  wordFromBitlist (bitlistFromWord w) = w

(*************************************************************************)
(* Comparisons                                                           *)
(*************************************************************************)

val mwordEq : ∀ α. MWORD α → MWORD α → BOOL
declare ocaml target_rep function mwordEq = 'Lem.word_equal'
let inline ~{ocaml} mwordEq = unsafe_structural_equality

instance ∀ α. (Eq (MWORD α))
  let = = mwordEq
  let <> w1 w2 = ¬ (mwordEq w1 w2)
end

val signedLess : ∀ α. MWORD α → MWORD α → BOOL

declare isabelle target_rep function signedLess = 'Word.word_sless'
declare hol target_rep function signedLess = 'words$word_lt'

val signedLessEq : ∀ α. MWORD α → MWORD α → BOOL

declare isabelle target_rep function signedLessEq = 'Word.word_sle'
declare hol target_rep function signedLessEq = 'words$word_le'

val unsignedLess : ∀ α. MWORD α → MWORD α → BOOL

declare isabelle target_rep function unsignedLess = infix '<'
declare hol target_rep function unsignedLess = 'words$word_lo'
declare ocaml target_rep function unsignedLess = 'Lem.unsignedLess'

val unsignedLessEq : ∀ α. MWORD α → MWORD α → BOOL

declare isabelle target_rep function unsignedLessEq = infix '\<le>'
declare hol target_rep function unsignedLessEq = 'words$word_ls'
declare ocaml target_rep function unsignedLessEq = 'Lem.unsignedLessEq'

let {ocaml} signedLess w1 w2 = (signedIntegerFromWord w1) < (signedIntegerFromWord w2)
```

```
let {ocaml} signedLessEq w1 w2 =  (signedIntegerFromWord w1) ≤ (signedIntegerFromWord w2)

(* Comparison tests are below, after the definition of wordFromInteger *)

(*****************************************************************)
(* Appending, splitting and probing words                        *)
(*****************************************************************)

val word_concat : ∀ α β γ. MWORD α → MWORD β → MWORD γ
declare hol target_rep function word_concat = 'words$word_concat'
declare isabelle target_rep function word_concat = 'Word.word_cat'
declare ocaml target_rep function word_concat = 'Lem.word_concat'

(* Note that we assume the result type has the correct size, especially  for Isabelle. *)
val word_extract : ∀ α β. NAT → NAT → MWORD α → MWORD β
declare hol target_rep function word_extract lo hi v = 'words$word_extract' hi lo v
declare isabelle target_rep function word_extract lo hi v = 'Word.slice' lo v
declare ocaml target_rep function word_extract = 'Lem.word_extract'

(* Needs to be in the prover because we'd end up with unknown sizes in the  types in Lem.*)
val word_update : ∀ α β. MWORD α → NAT → NAT → MWORD β → MWORD α
declare hol target_rep function word_update v lo hi w = 'words$bit_field_insert' hi lo w v
declare isabelle target_rep function word_update v lo hi w = 'Lem.word_update' v lo hi w
declare ocaml target_rep function word_update = 'Lem.word_update'

val setBit : ∀ α. MWORD α → NAT → BOOL → MWORD α

declare isabelle target_rep function setBit = 'Bits.set_bit'
declare hol target_rep function setBit w i b = '$:+' i b w
declare ocaml target_rep function setBit = 'Lem.word_setBit'

val getBit : ∀ α. MWORD α → NAT → BOOL

declare isabelle target_rep function getBit = 'Bits.test_bit'
declare hol target_rep function getBit w b = 'words$word_bit' b w
declare ocaml target_rep function getBit = 'Lem.word_getBit'

val msb : ∀ α. MWORD α → BOOL

declare isabelle target_rep function msb = 'Bits.msb'
declare hol target_rep function msb = 'words$word_msb'
declare ocaml target_rep function msb = 'Lem.word_msb'

val lsb : ∀ α. MWORD α → BOOL

declare isabelle target_rep function lsb = 'Bits.lsb'
declare hol target_rep function lsb = 'words$word_lsb'
declare ocaml target_rep function lsb = 'Lem.word_lsb'

assert {ocaml, hol, isabelle} extract_concat_test :
  let x : MWORD TY16 = wordFromNatural 1234 in
  word_concat ((word_extract 11 15 x) : MWORD TY5)
              ((word_concat ((word_extract 4 10 x) : MWORD TY7)
                            ((word_extract 0 3 x) : MWORD TY4)) : MWORD TY11)
    = x
assert {ocaml, hol, isabelle} update_test :
  let x : MWORD TY16 = wordFromNatural 1234 in
  let y : MWORD TY8 = wordFromNatural 41 in
```

word_update x 1 8 y = wordFromNatural 1106
assert {*ocaml*, *hol*, *isabelle*} *setBit_test*$_1$: setBit (wordFromNatural 12 : MWORD TY$_8$) 1 true = wordFromNatural 14

assert {*ocaml*, *hol*, *isabelle*} *setBit_test*$_2$: setBit (wordFromNatural 14 : MWORD TY$_8$) 1 false = wordFromNatural 12

assert {*ocaml*, *hol*, *isabelle*} *setBit_test*$_3$: setBit (wordFromNatural 2 : MWORD TY$_8$) 1 false = wordFromNatural 0

assert {*ocaml*, *hol*, *isabelle*} *getBit_test* : getBit (wordFromNatural 3 : MWORD TY$_8$) 1 = true

(* ********** *)
(* Bitwise operations, shifts, etc. *)
(* ********** *)

val *shiftLeft* : $\forall\ \alpha$. MWORD α → NAT → MWORD α

declare *isabelle* target_rep function shiftLeft = infix '<<'
declare *hol* target_rep function shiftLeft = 'words$word_lsl'
declare *ocaml* target_rep function shiftLeft = 'Lem.word_shiftLeft'

val *shiftRight* : $\forall\ \alpha$. MWORD α → NAT → MWORD α

declare *isabelle* target_rep function shiftRight = infix '>>'
declare *hol* target_rep function shiftRight = 'words$word_lsr'
declare *ocaml* target_rep function shiftRight = 'Lem.word_shiftRight'

val *arithShiftRight* : $\forall\ \alpha$. MWORD α → NAT → MWORD α

declare *isabelle* target_rep function arithShiftRight = infix '>>>'
declare *hol* target_rep function arithShiftRight = 'words$word_asr'
declare *ocaml* target_rep function arithShiftRight = 'Lem.word_arithShiftRight'

val *lAnd* : $\forall\ \alpha$. MWORD α → MWORD α → MWORD α

declare *isabelle* target_rep function lAnd = 'Bits.bitAND'
declare *hol* target_rep function lAnd = 'words$word_and'
declare *ocaml* target_rep function lAnd = 'Lem.word_and'

val *lOr* : $\forall\ \alpha$. MWORD α → MWORD α → MWORD α

declare *isabelle* target_rep function lOr = 'Bits.bitOR'
declare *hol* target_rep function lOr = 'words$word_or'
declare *ocaml* target_rep function lOr = 'Lem.word_or'

val *lXor* : $\forall\ \alpha$. MWORD α → MWORD α → MWORD α

declare *isabelle* target_rep function lXor = 'Bits.bitXOR'
declare *hol* target_rep function lXor = 'words$word_xor'
declare *ocaml* target_rep function lXor = 'Lem.word_xor'

val *lNot* : $\forall\ \alpha$. MWORD α → MWORD α

declare *isabelle* target_rep function lNot = 'Bits.bitNOT'
declare *hol* target_rep function lNot = 'words$word_1comp'
declare *ocaml* target_rep function lNot = 'Lem.word_not'

val *rotateRight* : $\forall\ \alpha$. NAT → MWORD α → MWORD α

declare *isabelle* target_rep function rotateRight = 'Word.word_rotr'

```
declare hol target_rep function rotateRight i w = 'words$word_ror' w i
declare ocaml target_rep function rotateRight = 'Lem.word_ror'

val rotateLeft : ∀ α. NAT → MWORD α → MWORD α

declare isabelle target_rep function rotateLeft = 'Word.word_rotl'
declare hol target_rep function rotateLeft i w = 'words$word_rol' w i
declare ocaml target_rep function rotateLeft = 'Lem.word_rol'

val zeroExtend : ∀ α β. Size β ⇒ MWORD α → MWORD β

declare isabelle target_rep function zeroExtend = 'Word.ucast'
declare hol target_rep function zeroExtend = 'words$w2w'
let {ocaml} zeroExtend x = wordFromNatural (naturalFromWord x)

val signExtend : ∀ α β. Size β ⇒ MWORD α → MWORD β

declare isabelle target_rep function signExtend = 'Word.scast'
declare hol target_rep function signExtend = 'words$sw2sw'
(* ocaml after definition for wordFromInteger *)

assert {ocaml, hol, isabelle} shift_test_1 : shiftLeft (wordFromNatural 5 : MWORD TY_8) 2 = wordFromNatural 20

assert {ocaml, hol, isabelle} shift_test_2 : shiftRight (wordFromNatural 5 : MWORD TY_8) 2 = wordFromNatural 1

assert {ocaml, hol, isabelle} shift_test_3 : shiftRight (wordFromNatural 129 : MWORD TY_8) 2 = wordFromNatural 32

assert {ocaml, hol, isabelle} shift_test_4 : arithShiftRight (wordFromNatural 129 : MWORD TY_8) 2 = wordFromNatural 224

assert {ocaml, hol, isabelle} and_test : lAnd (wordFromNatural 5) (wordFromNatural 36) = (wordFromNatural 4 :
MWORD TY_8)
assert {ocaml, hol, isabelle} or_test : lOr (wordFromNatural 5) (wordFromNatural 36) = (wordFromNatural 37 :
MWORD TY_8)
assert {ocaml, hol, isabelle} xor_test : lXor (wordFromNatural 5) (wordFromNatural 36) = (wordFromNatural 33 :
MWORD TY_8)
assert {ocaml, hol, isabelle} not_test : lNot (wordFromNatural 37) = (wordFromNatural 218 : MWORD TY_8)

assert {ocaml, hol, isabelle} rotateR_test : rotateRight 3 (wordFromNatural 37) = (wordFromNatural 164 :
MWORD TY_8)
assert {ocaml, hol, isabelle} rotateL_test : rotateLeft 3 (wordFromNatural 37) = (wordFromNatural 41 :
MWORD TY_8)
assert {ocaml, hol, isabelle} zext_test_0 : zeroExtend (wordFromNatural 130 : MWORD TY_8) = (wordFromNatural 130 :
MWORD TY_16)
assert {ocaml, hol, isabelle} zext_test_1 : zeroExtend (wordFromNatural 130 : MWORD TY_8) = (wordFromNatural 2 :
MWORD TY_7)

(* Sign extension tests are below, after the definition of wordFromInteger *)

(*****************************************************************)
(* Arithmetic                                                    *)
(*****************************************************************)

val plus : ∀ α. MWORD α → MWORD α → MWORD α

declare isabelle target_rep function plus = infix '+'
declare hol target_rep function plus = 'words$word_add'
declare ocaml target_rep function plus = 'Lem.word_plus'
```

```
val minus : ∀ α. MWORD α → MWORD α → MWORD α

declare isabelle target_rep function minus = infix '-'
declare hol target_rep function minus = 'words$word_sub'
declare ocaml target_rep function minus = 'Lem.word_minus'

val uminus : ∀ α. MWORD α → MWORD α

declare isabelle target_rep function uminus w = '-' w
declare hol target_rep function uminus = 'words$word_2comp'
declare ocaml target_rep function uminus = 'Lem.word_uminus'

val times : ∀ α. MWORD α → MWORD α → MWORD α

declare isabelle target_rep function times = infix '*'
declare hol target_rep function times = 'words$word_mul'
declare ocaml target_rep function times = 'Lem.word_times'

val unsignedDivide : ∀ α. MWORD α → MWORD α → MWORD α
val signedDivide : ∀ α. MWORD α → MWORD α → MWORD α

declare isabelle target_rep function unsignedDivide = infix 'div'
declare hol target_rep function unsignedDivide = 'words$word_div'
declare ocaml target_rep function unsignedDivide = 'Lem.word_udiv'
declare hol target_rep function signedDivide = 'words$word_quot'

let {isabelle, ocaml} signedDivide x y =
    if msb x then
      if msb y then unsignedDivide (uminus x) (uminus y)
      else uminus (unsignedDivide (uminus x) y)
   else if msb y then uminus (unsignedDivide x (uminus y))
      else unsignedDivide x y

val modulo : ∀ α. MWORD α → MWORD α → MWORD α

declare isabelle target_rep function modulo = infix 'mod'
declare hol target_rep function modulo = 'words$word_mod'
declare ocaml target_rep function modulo = 'Lem.word_mod'

(* Now we can define wordFromInteger for OCaml *)

let {ocaml} wordFromInteger i =
    if i < 0
   then uminus (wordFromNatural (naturalFromInteger (−i)))
   else wordFromNatural (naturalFromInteger i)

let {ocaml} signExtend x =  wordFromInteger (signedIntegerFromWord x)

val wordFromNumeral : ∀ α. Size α ⇒ NUMERAL → MWORD α

declare isabelle target_rep function wordFromNumeral n = ''n
declare hol target_rep function wordFromNumeral n = special "%ew" n
let inline {ocaml, coq} wordFromNumeral n = wordFromInteger (integerFromNumeral n)

instance ∀ α. Size α ⇒ (Numeral (MWORD α))
  let fromNumeral n =  wordFromNumeral n
end
```

assert {*ocaml*, *hol*, *isabelle*} *wordFromInteger_nat_test*$_1$: ((wordFromInteger 42) : MWORD TY$_8$) = ($0x2A$: MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *wordFromInteger_nat_test*$_2$: ((wordFromInteger (-42)) : MWORD TY$_8$) = uminus (wordFromN

assert {*ocaml*, *hol*, *isabelle*} *plus_test* : plus (wordFromInteger (-5) : MWORD TY$_8$) ($0b00000010$: MWORD TY$_8$) = wordFromInteger (-3)
assert {*ocaml*, *hol*, *isabelle*} *minus_test* : minus (wordFromInteger (-5) : MWORD TY$_8$) (wordFromNatural 2) = wordFromI

assert {*ocaml*, *hol*, *isabelle*} *times_test* : times (wordFromInteger (-5) : MWORD TY$_8$) (wordFromNatural 2) = wordFromIn

assert {*ocaml*, *hol*, *isabelle*} *udiv_test* : unsignedDivide (wordFromInteger (-5) : MWORD TY$_8$) (wordFromNatural 2) = wor

assert {*ocaml*, *hol*, *isabelle*} *sdiv_test* : signedDivide (wordFromInteger (-5) : MWORD TY$_8$) (wordFromNatural 2) = wordF

(* Comparison tests, which need wordFromInteger *)

assert {*ocaml*, *hol*, *isabelle*} *signedLess_test*$_1$: signedLess (wordFromInteger (-5)) ((wordFromInteger 3) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *signedLess_test*$_2$: signedLess (wordFromInteger 3) ((wordFromInteger 5) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *signedLess_test*$_3$: ¬ (signedLess (wordFromInteger 3) ((wordFromInteger 3) : MWORD TY$_8$))
assert {*ocaml*, *hol*, *isabelle*} *signedLessEq_test*$_1$: signedLessEq (wordFromInteger (-5)) ((wordFromInteger 3) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *signedLessEq_test*$_2$: signedLessEq (wordFromInteger 3) ((wordFromInteger 5) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *signedLessEq_test*$_3$: signedLessEq (wordFromInteger 3) ((wordFromInteger 3) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *unsignedLess_test*$_1$: unsignedLess (wordFromInteger 3) ((wordFromInteger 5) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *unsignedLess_test*$_2$: unsignedLess (wordFromInteger 3) ((wordFromInteger 255) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *unsignedLess_test*$_3$: ¬ (unsignedLess (wordFromInteger 255) ((wordFromInteger 255) : MWORD TY$_8$))
assert {*ocaml*, *hol*, *isabelle*} *unsignedLessEq_test*$_1$: unsignedLessEq (wordFromInteger 3) ((wordFromInteger 5) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *unsignedLessEq_test*$_2$: unsignedLessEq (wordFromInteger 3) ((wordFromInteger 255) : MWORD TY$_8$)
assert {*ocaml*, *hol*, *isabelle*} *unsignedLessEq_test*$_3$: unsignedLessEq (wordFromInteger 255) ((wordFromInteger 255) : MWORD TY$_8$)

(* sign extend tests *)

assert {*ocaml*, *hol*, *isabelle*} *sext_test*$_0$: signExtend (wordFromNatural 130 : MWORD TY$_8$) = (wordFromInteger (-126) : MWORD TY$_{16}$)
assert {*ocaml*, *hol*, *isabelle*} *sext_test*$_1$: signExtend (wordFromNatural 130 : MWORD TY$_8$) = (wordFromInteger 2 : MWORD TY$_7$)

27 Pervasives

declare {*isabelle*, *ocaml*, *hol*, *coq*} rename module = Lem_pervasives

include import *Basic_classes Bool Tuple Maybe Either Function Num Map Set List String Word Show*

import *Sorting Relation*

28 Pervasives_extra

declare {*isabelle*, *ocaml*, *hol*, *coq*} rename module = Lem_pervasives_extra

include import *Pervasives*
include import *Function_extra Maybe_extra Map_extra Num_extra Set_extra Set_helpers List_extra String_extra Assert_extra*