The Interchange Law in Application to Concurrent Programming — Mechanisation in Isabelle/HOL

Tony Hoare Bernard Möller Georg Struth Frank Zeyda

July 11, 2017

Abstract

Contents

1 Preliminaries

```
theory Preliminaries
imports Main Real Eisbach
  "~~/src/Tools/Adhoc_Overloading"
  "~~/src/HOL/Library/Monad_Syntax"
begin
```

1.1 Type Synonyms

Type synonym for homogeneous unary operators on a type 'a.

```
type_synonym 'a unop = "'a ⇒ 'a"
```

Type synonym for homogeneous binary operators on a type 'a.

```
type_synonym 'a binop = "'a ⇒ 'a ⇒ 'a"
end
```

2 The Option Monad: Supplement

```
theory Option_Monad
imports Preliminaries
  "~~/src/HOL/Library/Option_ord"
begin
```

While Isabelle/HOL already provides an encoding of the `option` type and monad, we include a few supplementary definitions and tactics here that are useful for readability and automatic proof.

2.1 Syntax and Definitions

The notation ⊥ is introduced for the constructor `None`.

```
notation None ("⊥")
```

We moreover define a `return` function for the option monad.

```
definition option_return :: "'a ⇒ 'a option" ("return") where
[simp]: "option_return x = Some x"
```

Note that `op` ≫= is already defined for type `option`.

2.2 Instantiations

More instantiations can be added here as desired.

```
instantiation option :: (zero) zero
begin
definition zero_option :: "'a option" where
[simp]: "zero_option = Some 0"
instance ..
end

instantiation option :: (one) one
begin
definition one_option :: "'a option" where
[simp]: "one_option = Some 1"
instance ..
end
```

2.3 Proof Support

Proof support for reasoning about `option` types.

Attribute used to collect definitional laws for operators.

```
named_theorems option_ops
  "definitial laws for operators of the option type/monad"
```

Tactic that facilitates proofs about `option` values.

```
lemmas split_option =
  split_option_all
  split_option_ex

method option_tac = (
```

```
  (atomize (full))?,
  (simp add: split_option option_ops),
  (clarsimp; simp?)?)
end
```

3 Strict Operators

```
theory Strict_Operators
imports Preliminaries Option_Monad
begin
```

Strict operators carry a subscript $_?$.

3.1 Equality

We define a strong notion of equality between undefined values.

```
fun lifted_equals :: "'a option ⇒ 'a option ⇒ bool" (infix "=?" 50) where
"Some x =? Some y ⟷ x = y" |
"Some x =? None ⟷ False" |
"None =? Some y ⟷ False" |
"None =? None ⟷ True"
```

3.2 Relational Operators

We also define lifted versions of arithmetic comparisons and subset.

```
fun lifted_leq :: "'a::ord option ⇒ 'a option ⇒ bool" (infix "≤?" 50) where
"Some x ≤? Some y ⟷ x ≤ y" |
"Some x ≤? None ⟷ False" |
"None ≤? Some y ⟷ True" |
"None ≤? None ⟷ True"

fun lifted_less :: "'a::ord option ⇒ 'a option ⇒ bool" (infix "<?" 50) where
"Some x <? Some y ⟷ x < y" |
"Some x <? None ⟷ False" |
"None <? Some y ⟷ True" |
"None <? None ⟷ False"
```

From Tony's note, it is not entirely clear to me how to define the operator below. It turns out though that `None ⊆? Some y` must be `True` in order to prove the ICL example (10). It is also not clear to me whether the result of `x ⊆? y` could be undefined or is always expected to be a simple value i.e. of type `bool`. Discuss this with Tony at a suitable time.

```
fun lifted_subset :: "'a set option ⇒ 'a set option ⇒ bool" (infix "⊆?" 50) where
"Some x ⊆? Some y ⟷ x ⊆ y" |
"Some x ⊆? None ⟷ (*True*) False" |
"None ⊆? Some y ⟷ (*True*) False" |
"None ⊆? None ⟷ True"
```

The above definitions coincide with the default ordering on `option`.

```
lemma lifted_leq_equiv_option_ord:
"op ≤? = op ≤"
apply (rule ext)+
apply (rename_tac x y)
apply (option_tac)
done

lemma lifted_less_equiv_option_ord:
"op <? = op <"
apply (rule ext)+
```

```
apply (rename_tac x y)
apply (option_tac)
done
```

3.3 Multiplication and Division

Multiplication and division of (possibly) undefined values are defined by way of monadic lifting, using Isabelle/HOL's built-in support for monad syntax.

```
definition lifted_times :: "'a::times option binop" (infixl "*?" 70) where
"x *? y = do {x' ← x; y' ← y; return (x' * y')}"

definition lifted_divide :: "'a::{divide, zero} option binop" (infixl "'/?" 70) where
"x /? y = do {x' ← x; y' ← y; if y' ≠ 0 then return (x' div y') else ⊥}"
```

3.4 Union and Disjoint Union

Ditto for union and disjoint union.

```
definition lifted_union :: "'a set option binop" (infixl "∪?" 70) where
"x ∪? y = do {x' ← x; y' ← y; return (x' ∪ y')}"

definition disjoint_union :: "'a set option binop" (infixl "⊕?" 70) where
"x ⊕? y = do {x' ← x; y' ← y; if x' ∩ y' = {} then return (x' ∪ y') else ⊥}"
```

We configure the above operators to be unfolded by `option_tac`.

```
declare lifted_times_def [option_ops]
declare lifted_divide_def [option_ops]
declare lifted_union_def [option_ops]
declare disjoint_union_def [option_ops]
end
```

4 Machine Numbers

```
theory Machine_Number
imports Preliminaries
begin
```

4.1 Type Class

Machine numbers are introduced via a type class `machine_number`. The class extends a linear order by including a constant `max_number` that yields the largest representable number.

```
class machine_number = linorder +
  fixes max_number :: "'a"
begin
```

All numbers less or equal to `max_number` are within range.

```
definition number_range :: "'a set" where
[simp]: "number_range = {x. x ≤ max_number}"
end
```

It is not difficult to prove that `number_range` is non-empty.

```
lemma ex_leq_max_number:
"∃x. x ≤ max_number"
apply (rule_tac x = "max_number" in exI)
apply (rule order_refl)
done

lemma ex_in_number_range:
"∃x. x ∈ number_range"
apply (clarsimp)
apply (rule ex_leq_max_number)
done
```

4.2 Type Definition

We furthermore introduce a sub-type for representable numbers.

```
typedef (overloaded)
  'a::machine_number machine_number = "number_range::'a set"
apply (rule ex_in_number_range)
done
```

The notation `MN(_)` is declared for the abstraction function.

```
notation Abs_machine_number ("MN'(_')")
```

The notation ⟦_⟧ is declared for the representation function.

```
notation Rep_machine_number ("⟦_⟧")

setup_lifting type_definition_machine_number
```

Proof Support

```
lemmas Rep_machine_number_inject_sym = sym [OF Rep_machine_number_inject]

declare Abs_machine_number_inverse
```

```
  [simplified number_range_def mem_Collect_eq, simp]

declare Rep_machine_number_inverse
  [simplified number_range_def mem_Collect_eq, simp]

declare Abs_machine_number_inject
  [simplified number_range_def mem_Collect_eq, simp]

declare Rep_machine_number_inject_sym
  [simplified number_range_def mem_Collect_eq, simp]
```

4.3 Instantiations

4.3.1 Linear Order

```
instantiation machine_number :: (machine_number) linorder
begin
definition less_eq_machine_number ::
  "'a machine_number ⇒ 'a machine_number ⇒ bool" where
[simp]: "less_eq_machine_number x y ⟷ ⟦x⟧ ≤ ⟦y⟧"

definition less_machine_number ::
  "'a machine_number ⇒ 'a machine_number ⇒ bool" where
[simp]: "less_machine_number x y ⟷ ⟦x⟧ < ⟦y⟧"
instance
apply (intro_classes)
apply (unfold less_eq_machine_number_def less_machine_number_def)
— Subgoal 1
apply (transfer')
apply (rule less_le_not_le)
— Subgoal 2
apply (transfer')
apply (rule order_refl)
— Subgoal 3
apply (transfer')
apply (erule order_trans)
apply (assumption)
— Subgoal 4
apply (transfer')
apply (erule antisym)
apply (assumption)
— Subgoal 5
apply (transfer')
apply (rule linear)
done
end
```

4.3.2 Arithmetic Operators

```
instantiation machine_number :: ("{machine_number, zero}") zero
begin
definition zero_machine_number :: "'a machine_number" where
[simp]: "zero_machine_number = MN(0)"
instance ..
end
```

```
instantiation machine_number :: ("{machine_number, one}") one
begin
definition one_machine_number :: "'a machine_number" where
[simp]: "one_machine_number = MN(1)"
instance ..
end

instantiation machine_number :: ("{machine_number, plus}") plus
begin
definition plus_machine_number :: "'a machine_number binop" where
[simp]: "plus_machine_number x y = MN(⟦x⟧ + ⟦y⟧)"
instance ..
end

instantiation machine_number :: ("{machine_number, minus}") minus
begin
definition minus_machine_number :: "'a machine_number binop" where
[simp]: "minus_machine_number x y = MN(⟦x⟧ - ⟦y⟧)"
instance ..
end

instantiation machine_number :: ("{machine_number, times}") times
begin
definition times_machine_number :: "'a machine_number binop" where
[simp]: "times_machine_number x y = MN(⟦x⟧ * ⟦y⟧)"
instance ..
end

instantiation machine_number :: ("{machine_number, divide}") divide
begin
definition divide_machine_number :: "'a machine_number binop" where
[simp]: "divide_machine_number x y = MN(⟦x⟧ div ⟦y⟧)"
instance ..
end
end
```

5 The Overflow Monad

```
theory Overflow_Monad
imports Machine_Number
  Preliminaries
begin
```

5.1 Type Definition

Any type with a linear order can be lifted into a type that includes ⊤.

```
datatype 'a::linorder overflow =
  Value "'a" |
  Overflow ("⊤")
```

5.2 Proof Support

Attribute used to collect definitional laws for operators.

```
named_theorems overflow_ops
  "definitial laws for operators of the overflow type/monad"

lemma split_overflow_all:
"(∀x. P x) = (P Overflow ∧ (∀x. P (Value x)))"
apply (safe)
— Subgoal 1
apply (clarsimp)
— Subgoal 2
apply (clarsimp)
— Subgoal 3
apply (case_tac x)
apply (simp_all)
done

lemma split_overflow_ex:
"(∃x. P x) = (P Overflow ∨ (∃x. P (Value x)))"
apply (safe)
— Subgoal 1
apply (case_tac x)
apply (simp_all) [2]
— Subgoal 2
apply (auto) [1]
— Subgoal 3
apply (auto) [1]
done

lemmas split_overflow =
  split_overflow_all
  split_overflow_ex
```

Tactic that facilitates proofs about the `overflow` type.

```
method overflow_tac = (
  (atomize (full))?,
  (simp add: split_overflow overflow_ops),
  (clarsimp; simp?)?)
```

5.3 Ordering Relation

Overflow (⊤) resides above any other value in the order.

```
instantiation overflow :: (linorder) linorder
begin
fun less_eq_overflow :: "'a overflow ⇒ 'a overflow ⇒ bool" where
"Value x ≤ Value y ⟷ x ≤ y" |
"Value x ≤ Overflow ⟷ True" |
"Overflow ≤ Value x ⟷ False" |
"Overflow ≤ Overflow ⟷ True"

fun less_overflow :: "'a overflow ⇒ 'a overflow ⇒ bool" where
"Value x < Value y ⟷ x < y" |
"Value x < Overflow ⟷ True" |
"Overflow < Value x ⟷ False" |
"Overflow < Overflow ⟷ False"
instance
apply (intro_classes)
— Subgoal 1
apply (overflow_tac)
apply (rule less_le_not_le)
— Subgoal 2
apply (overflow_tac)
— Subgoal 3
apply (overflow_tac)
— Subgoal 4
apply (overflow_tac)
— Subgoal 5
apply (overflow_tac)
done
end

instantiation overflow :: ("{linorder, zero}") zero
begin
definition zero_overflow :: "'a overflow" where
[simp]: "zero_overflow = Value 0"
instance ..
end

instantiation overflow :: ("{linorder, one}") one
begin
definition one_overflow :: "'a overflow" where
[simp]: "one_overflow = Value 1"
instance ..
end
```

5.4 Monadic Constructors

To support monadic syntax, we define the bind and return functions below.

```
primrec overflow_bind ::
  "'a::linorder overflow ⇒ ('a ⇒ 'b::linorder overflow) ⇒ 'b overflow" where
"overflow_bind (Overflow) f = Overflow" |
"overflow_bind (Value x)  f = f x"

adhoc_overloading bind overflow_bind
```

definition overflow_return :: "'a::linorder ⇒ 'a overflow" ("return") **where**
[simp]: "overflow_return x = Value x"

5.5 Lifted Operators

5.5.1 Generic Lifting

default_sort machine_number

Extended machine numbers are machine numbers that record an overflow.

type_synonym 'a machine_number_ext = "'a machine_number overflow"

translations
  (type) "'a machine_number_ext" ↽ (type) "'a machine_number overflow"

definition check_overflow :: "'a binop ⇒ 'a machine_number_ext binop" **where**
"check_overflow f x y = do {x' ← x; y' ← y;
  if (f ⟦x'⟧ ⟦y'⟧) ∈ number_range then return MN(f ⟦x'⟧ ⟦y'⟧) else ⊤}"

declare check_overflow_def [overflow_ops]

5.5.2 Concrete Operators

definition overflow_times ::
  "'a::{times, machine_number} machine_number_ext binop" (**infixl** "[*]" 70) **where**
[overflow_ops]: "overflow_times = check_overflow (op *)"

definition overflow_divide ::
  "'a::{divide, machine_number} machine_number_ext binop" (**infixl** "[div]" 70) **where**
[overflow_ops]: "overflow_divide = check_overflow (op div)"

default_sort type

5.6 Overflow Laws

lemma check_overflow_simps [simp]:
"check_overflow f x ⊤ = ⊤"
"check_overflow f ⊤ y = ⊤"
"check_overflow f (Value x') (Value y') =
  (if (f ⟦x'⟧ ⟦y'⟧) ≤ max_number then Value MN(f ⟦x'⟧ ⟦y'⟧) else ⊤)"
apply (unfold check_overflow_def)
apply (case_tac x; simp)
apply (case_tac y; simp)
apply (clarsimp)
done

5.7 Proof Experiments

instantiation nat :: machine_number
begin
definition max_number_nat :: "nat" **where**
"max_number_nat = 2 ^^ 31"
instance
apply (intro_classes)
done

```
end

lemma "⋀(x::nat machine_number_ext) y. x [*] y = y [*] x"
apply (overflow_tac)
apply (simp add: mult.commute)
done
end
```

6 Strict Operators

```
theory Strict_Operators
imports Preliminaries Option_Monad
begin
```

Strict operators carry a subscript $_?$.

6.1 Equality

We define a strong notion of equality between undefined values.

```
fun lifted_equals :: "'a option ⇒ 'a option ⇒ bool" (infix "=?" 50) where
"Some x =? Some y ⟷ x = y" |
"Some x =? None ⟷ False" |
"None =? Some y ⟷ False" |
"None =? None ⟷ True"
```

6.2 Relational Operators

We also define lifted versions of arithmetic comparisons and subset.

```
fun lifted_leq :: "'a::ord option ⇒ 'a option ⇒ bool" (infix "≤?" 50) where
"Some x ≤? Some y ⟷ x ≤ y" |
"Some x ≤? None ⟷ False" |
"None ≤? Some y ⟷ True" |
"None ≤? None ⟷ True"

fun lifted_less :: "'a::ord option ⇒ 'a option ⇒ bool" (infix "<?" 50) where
"Some x <? Some y ⟷ x < y" |
"Some x <? None ⟷ False" |
"None <? Some y ⟷ True" |
"None <? None ⟷ False"
```

From Tony's note, it is not entirely clear to me how to define the operator below. It turns out though that `None ⊆? Some y` must be `True` in order to prove the ICL example (10). It is also not clear to me whether the result of `x ⊆? y` could be undefined or is always expected to be a simple value i.e. of type `bool`. Discuss this with Tony at a suitable time.

```
fun lifted_subset :: "'a set option ⇒ 'a set option ⇒ bool" (infix "⊆?" 50) where
"Some x ⊆? Some y ⟷ x ⊆ y" |
"Some x ⊆? None ⟷ (*True*) False" |
"None ⊆? Some y ⟷ (*True*) False" |
"None ⊆? None ⟷ True"
```

The above definitions coincide with the default ordering on `option`.

```
lemma lifted_leq_equiv_option_ord:
"op ≤? = op ≤"
apply (rule ext)+
apply (rename_tac x y)
apply (option_tac)
done

lemma lifted_less_equiv_option_ord:
"op <? = op <"
apply (rule ext)+
```

```
apply (rename_tac x y)
apply (option_tac)
done
```

6.3 Multiplication and Division

Multiplication and division of (possibly) undefined values are defined by way of monadic lifting, using Isabelle/HOL's built-in support for monad syntax.

```
definition lifted_times :: "'a::times option binop" (infixl "*?" 70) where
"x *? y = do {x' ← x; y' ← y; return (x' * y')}"

definition lifted_divide :: "'a::{divide, zero} option binop" (infixl "'/?" 70) where
"x /? y = do {x' ← x; y' ← y; if y' ≠ 0 then return (x' div y') else ⊥}"
```

6.4 Union and Disjoint Union

Ditto for union and disjoint union.

```
definition lifted_union :: "'a set option binop" (infixl "∪?" 70) where
"x ∪? y = do {x' ← x; y' ← y; return (x' ∪ y')}"

definition disjoint_union :: "'a set option binop" (infixl "⊕?" 70) where
"x ⊕? y = do {x' ← x; y' ← y; if x' ∩ y' = {} then return (x' ∪ y') else ⊥}"
```

We configure the above operators to be unfolded by `option_tac`.

```
declare lifted_times_def [option_ops]
declare lifted_divide_def [option_ops]
declare lifted_union_def [option_ops]
declare disjoint_union_def [option_ops]
end
```

7 Partiality

```
theory Partiality
imports Preliminaries
  "~~/src/HOL/Library/Monad_Syntax"
begin
```

Our construction here adds a distinct ⊥ and ⊤ element to some type.

7.1 Type Definition

We define a datatype `'a partial` to lift values into 'extended values'.

```
datatype 'a partial =
  Bottom ("⊥") | Value "'a" | Top ("⊤")
```

7.2 Proof Support

Tactic that facilitates proofs about the `partial` type.

```
named_theorems partial_ops
  "definitional theorems for operators on the type partial"

lemma partial_split_all:
"(∀x::'a partial. P x) = (P Bottom ∧ P Top ∧ (∀x::'a. P (Value x)))"
apply (safe; simp?)
apply (case_tac x)
apply (simp_all)
done

lemma partial_split_ex:
"(∃x::'a partial. P x) = (P Bottom ∨ P Top ∨ (∃x::'a. P (Value x)))"
apply (safe; simp?)
apply (case_tac x)
apply (simp_all) [3]
apply (auto)
done

lemmas partial_split_laws =
  partial_split_all
  partial_split_ex

method partial_tac = (
  (atomize (full))?,
  (simp add: partial_split_laws partial_ops)?,
  (clarsimp; simp?)?)
```

7.3 Monadic Constructors

We have strictness in both ⊥ and ⊤.

```
primrec partial_bind ::
  "'a partial ⇒ ('a ⇒ 'b partial) ⇒ 'b partial" where
"partial_bind (Bottom) f = Bottom" |
"partial_bind (Value x) f = f x" |
"partial_bind (Top) f = Top"
```

adhoc_overloading bind partial_bind

```
definition partial_return :: "'a  ⇒ 'a partial" ("return") where
[simp]: "partial_return x = Value x"
```

7.4 Lifting Functors

```
fun lift_unop :: "('a ⇒ 'b) ⇒ ('a partial ⇒ 'b partial)" where
"lift_unop f Bottom = Bottom" |
"lift_unop f (Value x) = Value (f x)" |
"lift_unop f Top = Top"

fun lift_binop ::
  "('a ⇒ 'b ⇒ 'c) ⇒ ('a partial ⇒ 'b partial ⇒ 'c partial)" where
"lift_binop f Bottom Bottom = Bottom" |
"lift_binop f Bottom (Value y) = Bottom" |
"lift_binop f Bottom Top = Bottom" |
"lift_binop f (Value x) Bottom = Bottom" |
"lift_binop f (Value x) (Value y) = Value (f x y)" |
"lift_binop f (Value x) Top = Top" |
"lift_binop f Top Bottom = Bottom" |
"lift_binop f Top (Value y) = Top" |
"lift_binop f Top Top = Top"
```

7.5 Ordering Relation

```
primrec partial_ord :: "'a partial ⇒ nat" where
"partial_ord Bottom = 0" |
"partial_ord (Value x) = 1" |
"partial_ord Top = 2"

instantiation partial :: (ord) ord
begin
fun less_eq_partial :: "'a partial ⇒ 'a partial ⇒ bool" where
"(Value x) ≤ (Value y) ⟷ x ≤ y" |
"a ≤ b ⟷ (partial_ord a) ≤ (partial_ord b)"

fun less_partial :: "'a partial ⇒ 'a partial ⇒ bool" where
"(Value x) < (Value y) ⟷ x < y" |
"a < b ⟷ (partial_ord a) < (partial_ord b)"
instance ..
end
```

7.6 Class Instantiations

7.6.1 Preorder

```
instance partial :: (preorder) preorder
apply (intro_classes)
— Subgoal 1
apply (partial_tac)
apply (rule less_le_not_le)
— Subgoal 2
apply (partial_tac)
— Subgoal 3
apply (partial_tac)
```

```
apply (erule order_trans)
apply (assumption)
done
```

7.6.2 Partial Order

```
instance partial :: (order) order
apply (intro_classes)
apply (partial_tac)
done
```

7.6.3 Linear Order

```
instance partial :: (linorder) linorder
apply (intro_classes)
apply (partial_tac)
done
```

7.6.4 Lattice

```
instantiation partial :: (type) bot
begin
definition bot_partial :: "'a partial" where
[partial_ops]: "bot_partial = Bottom"
instance ..
end

instantiation partial :: (type) top
begin
definition top_partial :: "'a partial" where
[partial_ops]: "top_partial = Top"
instance ..
end

notation inf (infixl "⊓" 70)
notation sup (infixl "⊔" 65)

instantiation partial :: (lattice) lattice
begin
fun inf_partial :: "'a partial ⇒ 'a partial ⇒ 'a partial" where
"Bottom ⊓ Bottom = Bottom" |
"Bottom ⊓ (Value y) = Bottom" |
"Bottom ⊓ Top = Bottom" |
"(Value x) ⊓ Bottom = Bottom" |
"(Value x) ⊓ (Value y) = Value (x ⊓ y)" |
"(Value x) ⊓ Top = (Value x)" |
"Top ⊓ Bottom = Bottom" |
"Top ⊓ Value y = Value y" |
"Top ⊓ Top = Top"

fun sup_partial :: "'a partial ⇒ 'a partial ⇒ 'a partial" where
"Bottom ⊔ Bottom = Bottom" |
"Bottom ⊔ (Value y) = (Value y)" |
"Bottom ⊔ Top = Top" |
"(Value x) ⊔ Bottom = (Value x)" |
"(Value x) ⊔ (Value y) = Value (x ⊔ y)" |
```

```
"(Value x) ⊔ Top = Top" |
"Top ⊔ Bottom = Top" |
"Top ⊔ (Value y) = Top" |
"Top ⊔ Top = Top"
instance
apply (intro_classes)
— Subgoal 1
apply (partial_tac)
— Subgoal 2
apply (partial_tac)
— Subgoal 3
apply (partial_tac)
— Subgoal 4
apply (partial_tac)
— Subgoal 5
apply (partial_tac)
— Subgoal 6
apply (partial_tac)
done
end

lemma partial_ord_inf_lemma [simp]:
"∀a b. partial_ord (a ⊓ b) = min (partial_ord a) (partial_ord b)"
apply (partial_tac)
done

lemma partial_ord_sup_lemma [simp]:
"∀a b. partial_ord (a ⊔ b) = max (partial_ord a) (partial_ord b)"
apply (partial_tac)
done
```

7.6.5 Complete Lattice

```
instantiation partial :: (complete_lattice) complete_lattice
begin
definition Inf_partial :: "'a partial set ⇒ 'a partial" where
[partial_ops]:
"Inf_partial xs =
  (if Bottom ∈ xs then Bottom else
    let values = {x. Value x ∈ xs} in
      if values = {} then Top else Value (Inf values))"

definition Sup_partial :: "'a partial set ⇒ 'a partial" where
[partial_ops]:
"Sup_partial xs =
  (if Top ∈ xs then Top else
    let values = {x. Value x ∈ xs} in
      if values = {} then Bottom else Value (Sup values))"
instance
apply (intro_classes)
— Subgoal 1
apply (partial_tac)
apply (simp add: Inf_lower)
— Subgoal 2
apply (partial_tac)
apply (metis Inf_greatest mem_Collect_eq)
```

— Subgoal 3
apply `(partial_tac)`
apply `(simp add: Sup_upper)`
— Subgoal 4
apply `(partial_tac)`
apply `(metis Sup_least mem_Collect_eq)`
— Subgoal 5
apply `(partial_tac)`
— Subgoal 6
apply `(partial_tac)`
done
end
end

8 ICL Examples

```
theory ICL_Examples
imports Main Real Strict_Operators Overflow_Monad
begin
```

```
declare [[syntax_ambiguity_warning=false]]
```

We are going to use the | symbol for parallel composition.

```
no_notation (ASCII)
  disj  (infixr "|" 30)
```

8.1 Locale Definitions

In this section, we encapsulate the interchange law as an Isabelle locale. This gives us an elegant way to formulate conjectures that particular types, orderings, and operator pairs fulfill the interchange law. It also aids in structuring proofs. We define two locales here: one to introduce the notion of order (which has to be a preorder) and another, extending the former, to introduce both operators and interchange law as an assumption.

8.1.1 Locale: preorder

The underlying relation has to be a preorder. Our definition of preorder is, however, deliberately weaker than Isabelle/HOL's definition captured by the `ordering` locale. That is, we shall not require the assumption `ordering ?less_eq ?less ⟹ ?less ?a ?b = (?less_eq ?a ?b ∧ ?a ≠ ?b)`. Moreover, for interpretation we only have to provide the $\leq$ operator in our treatment and not `<` as well.

```
locale preorder =
  fixes type :: "'a itself"
  fixes less_eq :: "'a ⇒ 'a ⇒ bool" (infix "≤" 50)
  assumes refl: "x ≤ x"
  assumes trans: "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
begin
```

Equivalence of elements is defined in terms of mutual less-or-equals.

```
definition equiv :: "'a ⇒ 'a ⇒ bool" (infix "≡" 50) where
"x ≡ y ⟷ x ≤ y ∧ y ≤ x"
```

We prove that $\equiv$ is indeed an equivalence relation.

```
lemma equiv_refl:
"x ≡ x"
apply (unfold equiv_def)
apply (clarsimp)
apply (rule local.refl)
done

lemma equiv_sym:
"x ≡ y ⟹ y ≡ x"
apply (unfold equiv_def)
apply (clarsimp)
done

lemma equiv_trans:
```

```
"x ≡ y ⟹ y ≡ z ⟹ x ≡ z"
apply (unfold equiv_def)
apply (clarsimp)
apply (rule conjI)
using local.trans apply (blast)
using local.trans apply (blast)
done
```

The following anti-symmetry law holds by definition of equivalence.

```
lemma antisym:
"x ≤ y ⟹ y ≤ x ⟹ x ≡ y"
apply (unfold equiv_def)
apply (clarsimp)
done
end
```

Next, we prove several useful interpretations of `ICL_Examples.preorders`. Due to the structuring mechanism of (sub)locales, we are later able to reuse those instantiation proofs i.e. when interpreting of the `iclaw` locale defined in the sequel.

```
interpretation preorder_eq:
  preorder "TYPE('a)" "(op =)"
apply (unfold_locales)
apply (simp_all)
done

interpretation preorder_leq:
  preorder "TYPE('a::preorder)" "(op ≤)"
apply (unfold_locales)
apply (rule order_refl)
apply (erule order_trans; assumption)
done

interpretation preorder_subset:
  preorder "TYPE('a set)" "(op ⊆)"
apply (unfold_locales)
done

interpretation preorder_option_eq:
  preorder "TYPE('a option)" "(op =?)"
apply (unfold_locales)
apply (option_tac)+
done

interpretation preorder_option_leq:
  preorder "TYPE('a::preorder option)" "(op ≤?)"
apply (unfold_locales)
apply (option_tac)
apply (option_tac)
using order_trans apply (auto)
done

interpretation preorder_option_subset:
  preorder "TYPE('a set option)" "(op ⊆?)"
apply (unfold_locales)
apply (option_tac)
```

```
apply (option_tac)
apply (blast)
done
```

Make the above instantiation lemmas automatic simplifications.

```
declare preorder_eq.preorder_axioms [simp]
declare preorder_leq.preorder_axioms [simp]
declare preorder_option_eq.preorder_axioms [simp]
declare preorder_option_leq.preorder_axioms [simp]
```

8.1.2 Locale: iclaw

We are ready now to define the iclaw locale as an extension of the preorder locale. The interchange law is encapsulated as the single assumption of that locale. Instantiations will have to prove this assumption and thereby show that the interchange law holds for a particular given type, ordering relation, and binary operator pair.

```
locale iclaw = preorder +
  fixes seq_op :: "'a binop" (infixr ";" 110)
  fixes par_op :: "'a binop" (infixr "|" 100)
— 1. Note: the general shape of the interchange law.
  assumes interchange_law: "(p | r) ; (q | s) ≤ (p ; q) | (r ; s)"
```

8.2 ICL Interpretations

In this section, we prove the various instantiations of the interchange law in **Part 1** of the paper.

8.2.1 Arithmetic: addition (+) and subtraction (-) of numbers.

This is proved for the types int, rat and real.

— Note that the law does not hold for type nat.

```
interpretation icl_plus_minus_nat:
  iclaw "TYPE(nat)" "op =" "op +" "op -"
apply (unfold_locales)
apply (linarith?)
oops

interpretation icl_plus_minus_int:
  iclaw "TYPE(int)" "op =" "op +" "op -"
apply (unfold_locales)
apply (linarith)
done

interpretation icl_plus_minus_rat:
  iclaw "TYPE(rat)" "op =" "op +" "op -"
apply (unfold_locales)
apply (linarith)
done

interpretation icl_plus_minus_real:
  iclaw "TYPE(real)" "op =" "op +" "op -"
apply (unfold_locales)
```

```
apply (linarith)
done
```

8.2.2 Arithmetic: multiplication (x) and division (/) of numbers.

This is proved for the types `rat`, `real`, and option types thereof.

```
interpretation icl_mult_div_rat:
  iclaw "TYPE(rat)" "op =" "op *" "op /"
apply (unfold_locales)
apply (simp)
done

interpretation icl_mult_div_real:
  iclaw "TYPE(real)" "op =" "op *" "op /"
apply (unfold_locales)
apply (simp)
done
```

The `option_tac` tactic makes the two proofs below very easy.

```
interpretation icl_mult_div_rat_strong:
  iclaw "TYPE(rat option)" "op =?" "op *?" "op /?"
apply (unfold_locales)
apply (option_tac)
done

interpretation icl_mult_div_real_strong:
  iclaw "TYPE(real option)" "op =?" "op *?" "op /?"
apply (unfold_locales)
apply (option_tac)
done
```

Theorem 1 is true for any `field` in general, ...

```
lemma Theorem1_field:
fixes p :: "'a::field"
fixes q :: "'a::field"
shows "(p / q) * q = (p * q) / q"
using times_divide_eq_left by (blast)
```

... and `rational` and `real` numbers in particular.

```
lemma Theorem1_rat:
fixes p :: "rat"
fixes q :: "rat"
shows "(p / q) * q = (p * q) / q"
apply (rule Theorem1_field)
done

lemma Theorem1_real:
fixes p :: "real"
fixes q :: "real"
shows "(p / q) * q = (p * q) / q"
apply (rule Theorem1_field)
done
```

8.2.3 Natural numbers: multiplication (x) and truncated division (-:-)

We note that `x div y` is used in Isabelle for truncated division.

We first prove the lemma below which is also described in the paper.

```
lemma trunc_div_mult_leq:
fixes p :: "nat"
fixes q :: "nat"
— The assumption q > 0 is not needed because x div 0 = 0.
shows "(p div q) * q ≤ (p * q) div q"
apply (case_tac "q > 0")
apply (metis div_mult_self_is_m mult.commute split_div_lemma)
apply (simp)
done
```

We note that Isabelle/HOL defines `x div 0 = 0`. Hence we can prove the law even in HOL's weak treatment of undefinedness, as well as the stronger one.

```
interpretation icl_mult_trunc_div_nat:
  iclaw "TYPE(nat)" "op ≤" "op *" "op div"
apply (unfold_locales)
apply (case_tac "r = 0"; simp_all)
apply (case_tac "s = 0"; simp_all)
apply (subgoal_tac "(p div r) * (q div s) * (r * s) ≤ p * q")
apply (metis div_le_mono div_mult_self_is_m nat_0_less_mult_iff)
apply (unfold semiring_normalization_rules(13))
apply (metis div_mult_self_is_m mult_le_mono trunc_div_mult_leq)
done

interpretation icl_mult_trunc_div_nat_strong:
  iclaw "TYPE(nat option)" "op ≤?" "op *?" "op /?"
apply (unfold_locales)
apply (option_tac)
apply (subgoal_tac "(p div r) * (q div s) * (r * s) ≤ p * q")
apply (metis div_le_mono div_mult_self_is_m nat_0_less_mult_iff)
apply (unfold semiring_normalization_rules(13))
apply (metis div_mult_self_is_m mult_le_mono trunc_div_mult_leq)
done
```

8.2.4 Propositional calculus: conjunction (∧) and implication (⇒).

TO: Implication `p ⇒ q` is defined in the usual way as `¬p ∨ q`.

We can easily verify the definition of implication.

```
lemma "(p ⟶ q) ≡ (¬ p ∨ q)"
apply (simp)
done
```

We note that ⊢ is encoded by object-logic implication (⟶).

```
definition turnstile :: "bool ⇒ bool ⇒ bool" (infix "⊢" 50) where
[iff]: "turnstile p q ≡ p ⟶ q"

interpretation icl_imp_conj:
  iclaw "TYPE(bool)" "op ⟶" "op ∧" "op ⊢"
apply (unfold_locales)
```

```
apply (auto)
done
```

8.2.5 Boolean Algebra: conjunction (∧) and disjunction (∨).

Numerical value of a boolean value.

```
definition valOfBool :: "bool ⇒ nat" where
"valOfBool p = (if p then 1 else 0)"
```

Order on boolean values induced by the above.

```
definition numOrdBool :: "bool ⇒ bool ⇒ bool" (infix "⊢" 50) where
"numOrdBool p q ⟷ (valOfBool p) ≤ (valOfBool q)"
```

We can easily show that the numerical order is just implication.

```
lemma numOrdBool_is_imp [simp]:
"(numOrdBool p q) = (p ⟶ q)"
apply (unfold numOrdBool_def valOfBool_def)
apply (induct_tac p; induct_tac q)
apply (simp_all)
done
```

Note that ';' is disjunction and '|' is conjunction.

```
interpretation icl_boolean_algebra:
  iclaw "TYPE(bool)" "numOrdBool" "op ∨" "op ∧"
apply (unfold_locales)
apply (unfold numOrdBool_is_imp)
apply (auto)
done
```

8.2.6 Self-interchanging operators: +, *, ∨, ∧.

For convenience, we define a locale for self-interchanging operators.

```
locale self_iclaw =
  iclaw "type" "op =" "self_op" "self_op"
  for type :: "'a itself" and self_op  :: "'a binop"
```

We next introduce (separate) locales to capture associativity, commutativity and existence of units for some binary operator. We use a bold circle (**○**) not to clash with the Isabelle/HOL's symbol (○) for functional composition.

```
locale associative =
  fixes operator :: "'a binop" (infix "○" 100)
  assumes assoc: "x ○ (y ○ z) = (x ○ y) ○ z"

locale commutative =
  fixes operator :: "'a binop" (infix "○" 100)
  assumes comm: "x ○ y = y ○ x"

locale has_unit =
  fixes operator :: "'a binop" (infix "○" 100)
  fixes unit :: "'a" ("1")
  assumes left_unit [simp]: "1 ○ x = x"
  assumes right_unit [simp]: "x ○ 1 = x"
```

```
lemma assoc_comm_imp_self_iclaw:
"(associative bop ∧ commutative bop) ⟹ (self_iclaw bop)"
apply (standard)
apply (unfold associative_def commutative_def)
apply (clarify)
apply (auto)
done

lemma self_iclaw_unit_imp_assoc:
"(self_iclaw bop) ∧ (has_unit bop one) ⟹ associative bop"
apply (standard)
apply (unfold self_iclaw_def iclaw_def iclaw_axioms_def)
apply (clarsimp)
apply (drule_tac x = "x" in spec)
apply (drule_tac x = "one" in spec)
apply (drule_tac x = "y" in spec)
apply (drule_tac x = "z" in spec)
apply (simp add: has_unit_def)
done

lemma self_iclaw_unit_imp_comm:
"(self_iclaw bop) ∧ (has_unit bop one) ⟹ commutative bop"
apply (standard)
apply (unfold self_iclaw_def iclaw_def iclaw_axioms_def)
apply (clarsimp)
apply (drule_tac x = "one" in spec)
apply (drule_tac x = "x" in spec)
apply (drule_tac x = "y" in spec)
apply (drule_tac x = "one" in spec)
apply (simp add: has_unit_def)
done
```

Lastly, we prove the self-interchange law for the four operators.

```
interpretation self_icl_plus:
  self_iclaw "TYPE('a::comm_monoid_add)" "op +"
apply (rule assoc_comm_imp_self_iclaw)
apply (rule conjI)
— Subgoal 1
apply (unfold associative_def)
apply (simp add: add.assoc)
— Subgoal 2
apply (unfold commutative_def)
apply (simp add: add.commute)
done

interpretation self_icl_mult:
  self_iclaw "TYPE('a::comm_monoid_mult)" "op *"
apply (rule assoc_comm_imp_self_iclaw)
apply (rule conjI)
— Subgoal 1
apply (unfold associative_def)
apply (simp add: mult.assoc)
— Subgoal 2
apply (unfold commutative_def)
apply (simp add: mult.commute)
```

done

interpretation `self_icl_conj:`
```
  self_iclaw "TYPE(bool)" "op ∧"
```
apply `(rule assoc_comm_imp_self_iclaw)`
apply `(rule conjI)`
— Subgoal 1
apply `(standard) [1]`
apply `(blast)`
— Subgoal 2
apply `(standard) [1]`
apply `(blast)`
done

interpretation `self_icl_disj:`
```
  self_iclaw "TYPE(bool)" "op ∨"
```
apply `(rule assoc_comm_imp_self_iclaw)`
apply `(rule conjI)`
— Subgoal 1
apply `(standard) [1]`
apply `(blast)`
— Subgoal 2
apply `(standard) [1]`
apply `(blast)`
done

8.2.7 Computer arithmetic: Overflow ($\top$).

default_sort `machine_number`

To encode the outcome of a computer calculation, we firstly introduce a new type `'a comparith`. Such is an `option` type over the machine number type `'a machine_number_ext`. The latter is introduced in the theory `Overflow_Monad`. The order of nesting indeed ensures that we obtain the correct strictness properties with respect to $\bot$ and $\top$, that is $\bot$ dominates over $\top$.

type_synonym `'a comparith = "('a machine_number_ext) option"`

The definition operators for the `comparith` type follows next.

abbreviation `comparith_top :: "'a comparith" ("⊤")` **where**
`"comparith_top ≡ Some ⊤"`

definition `comparith_times ::`
```
  "'a::{times,machine_number} comparith binop" (infixl "*M" 70) where
[option_ops]: "x *M y = do {x' ← x; y' ← y; return (x' [*] y')}"
```

definition `comparith_divide ::`
```
  "'a::{zero,divide,machine_number} comparith binop" (infixl "'/M" 70) where
[option_ops]:
"x /M y = do {x' ← x; y' ← y; if y' ≠ 0 then return (x' [div] y') else ⊥}"
```

Cancellation Laws

lemma `Section_8_cancellation_law1a:`
```
"⋀ (p::nat machine_number_ext) (q::nat machine_number_ext).
  q ≠ 0 ⟹ p ≤ (p [*] q) [div] q"
```
apply `(overflow_tac)`

```
done

lemma Section_8_cancellation_law1b:
"⋀(p::nat comparith) (q::nat comparith).
  q ≠ 0 ⟹ q ≠ ⊥ ⟹ p ≤ (p *_M q) /_M q"
apply (option_tac)
apply (overflow_tac)
done

lemma Section_8_cancellation_law2a:
"⋀(p::nat option) (q::nat option).
  (p /_? q) *_? q ≤ p"
apply (option_tac)
apply (metis mult.commute split_div_lemma)
done

lemma Section_8_cancellation_law2b:
"⋀(p::nat comparith) (q::nat comparith).
  q ≠ ⊤ ⟹ (p /_M q) *_M q ≤ p"
apply (option_tac)
apply (overflow_tac)
apply (transfer)
apply (clarsimp; safe)
— Subgoal 1
apply (metis mult.commute split_div_lemma)
— Subgoal 2
using div_le_dividend dual_order.trans apply (blast)
— Subgoal 3
apply (erule contrapos_np)
apply (clarsimp)
apply (metis dual_order.trans mult.commute split_div_lemma)
done
```

Interchange Law

```
lemma overflow_times_neq_Value_MN_0 [rule_format]:
"⋀(x::nat machine_number_ext) (y::nat machine_number_ext).
 x ≠ Value MN(0) ⟹
 y ≠ Value MN(0) ⟹ x [*] y ≠ Value MN(0)"
apply (overflow_tac)
done

interpretation icl_mult_trunc_div_nat_overflow:
  iclaw "TYPE(nat comparith)" "op ≤" "op *_M" "op /_M"
apply (unfold_locales)
apply (option_tac)
apply (simp add: overflow_times_neq_Value_MN_0)
apply (unfold overflow_times_def overflow_divide_def)
apply (thin_tac "r ≠ Value MN(0)")
apply (thin_tac "s ≠ Value MN(0)")
apply (overflow_tac)
apply (transfer)
apply (clarsimp)
apply (safe)
using icl_mult_trunc_div_nat.interchange_law apply (blast)
using div_le_dividend dual_order.trans apply (blast)
```

```
apply (meson dual_order.trans icl_mult_trunc_div_nat.interchange_law)
using div_le_dividend dual_order.trans apply (blast)
done

default_sort type
```

8.2.8 Note: Partial operators.

Partial operators are formalised in a separate theory `Partiality`.

8.2.9 Sets: union (∪) and disjoint union (+) of sets, ordered by inclusion ⊆.

```
interpretation preorder_option_subset:
  iclaw "TYPE('a set option)" "(op ⊆?)" "op ⊕?" "op ∪?"
apply (unfold_locales)
apply (rename_tac p q r s)
apply (option_tac)
apply (auto)
— Cannot be proved unless we change the definition of op ⊆?!
oops

interpretation disjoint_union_unit:
  has_unit "op ⊕?" "Some {}"
apply (unfold_locales)
apply (option_tac)
apply (option_tac)
done

lemma [rule_format]:
"∀p. p = p ⊕? p ⟷ (p = ⊥ ∨ p = Some {})"
apply (option_tac)
done
end
```