The Interchange Law: A Principle of Concurrent Programming — Mechanisation in Isabelle/HOL

Tony Hoare Bernard Möller Georg Struth Frank Zeyda

August 28, 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 relational operators on a type `'a`.

```
type_synonym 'a relop = "'a ⇒ 'a ⇒ bool"
```

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"
```

1.2 Lattice Syntax

We use the constants below for ad hoc overloading to avoid ambiguities.

```
consts global_bot :: "'a" ("⊥")
consts global_top :: "'a" ("⊤")
```

Declaration of global notations for lattice operators.

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

notation
  Inf ("⨅") and
  Sup ("⨆")
```

1.3 Reverse Implication

```
abbreviation (input) rimplies :: "[bool, bool] ⇒ bool" (infixr "⟵" 25)
where "Q ⟵ P ≡ P ⟶ Q"
```

1.4 Monad Syntax

We use the constant below for ad hoc overloading to avoid ambiguities.

```
consts return :: "'a ⇒ 'b" ("return")
```

1.5 Equivalence Operator

Equivalence is introduced by extending the type class `ord`.

```
definition (in ord) equiv :: "'a ⇒ 'a ⇒ bool" (infix "≅" 50) where
[iff]: "x ≅ y ⟷ x ≤ y ∧ y ≤ x"

context preorder
begin
```

```
lemma equiv_relf:
"x ≅ x"
apply (clarsimp)
done

lemma equiv_sym:
"x ≅ y ⟹ y ≅ x"
apply (clarsimp)
done

lemma equiv_trans:
"x ≅ y ⟹ y ≅ z ⟹ x ≅ z"
apply (safe)
apply (erule order_trans; assumption)
apply (erule order_trans; assumption)
done
end
end
```

2 The Option Monad

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

Whilst 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 later on.

2.1 Syntax and Definitions

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

```
adhoc_overloading global_bot None
```

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

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

adhoc_overloading return option_return
```

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

2.2 Instantiations

More instantiations can be added here as we desire.

```
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

Attribute used to collect definitional laws for operators.

```
named_theorems option_ops "definitional laws for operators on option values"
```

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 ICL
begin
```

All strict operators (on `option` types) carry a subscript $_?$.

3.1 Equality

We define a strong notion of equality between undefined values.

```
fun equals_option :: "'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"
```

The above indeed coincides with HOL equality.

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

3.2 Relational Operators

We also define lifted versions of the default orders ≤ and <.

```
fun leq_option :: "'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 less_option :: "'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"
```

Likewise, we can prove these correspond to HOL's default lifted order.

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

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

Lastly, we lift subset inclusion into the `option` type.

From Tony's note, it is not entirely clear to me how to define this operator It turns out that `None ⊆? Some y` has to be `True` in order to prove the ICL example (10). Besides, may the result of `x ⊆? y` be undefined too? Or do we always expected a simple `boolean` value when applying lifted relational operators? Discuss this with Tony and Georg at a suitable moment.

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

3.3 Generic Lifting

We use the constant below for ad hoc overloading to avoid ambiguities.

```
consts lift_option :: "'a ⇒ 'b" ("_↑?" [1000] 1000)

definition ulift_option ::
  "('a ⇒ 'b) ⇒ ('a option ⇒ 'b option)" where
"ulift_option f x = do {x' ← x; return (f x')}"

definition blift_option ::
  "('a ⇒ 'b ⇒ 'c) ⇒
   ('a option ⇒ 'b option ⇒ 'c option)" where
"blift_option f x y = do {x' ← x; y' ← y; return (f x' y')}"

adhoc_overloading lift_option ulift_option
adhoc_overloading lift_option blift_option
```

Note that we do not add the above operators to `option_ops`.

```
lemma ulift_option_simps [simp]:
"ulift_option f ⊥ = ⊥"
"ulift_option f (Some x) = Some (f x)"
apply (unfold ulift_option_def)
apply (simp_all)
done

lemma blift_option_simps [simp]:
"blift_option f x ⊥ = ⊥"
"blift_option f ⊥ y = ⊥"
"blift_option f (Some x') (Some y') = Some (f x' y')"
apply (unfold blift_option_def)
apply (simp_all)
done
```

3.4 Lifted Operators

Addition and Subtraction

```
definition plus_option :: "'a::plus option binop" (infixl "+?" 70) where
"(op +?) = (op +)↑?"

definition minus_option :: "'a::minus option binop" (infixl "-?" 70) where
"(op -?) = (op -)↑?"
```

Multiplication and Division

```
definition times_option :: "'a::times option binop" (infixl "*?" 70) where
"(op *?) = (op *)↑?"

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

Union and Disjoint Union

```
definition union_option :: "'a set option binop" (infixl "∪?" 70) where
"(op ∪? ) = (op ∪)↑?"

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 ⊥}"
```

Proof Support

```
declare plus_option_def [option_ops]
declare minus_option_def [option_ops]
declare times_option_def [option_ops]
declare divide_option_def [option_ops]
declare union_option_def [option_ops]
declare disjoint_union_def [option_ops]
```

3.5 Supplementary Laws

```
lemma div_by_1_option [simp]:
fixes a :: "'a::semidom_divide option"
shows "a /? 1 = a"
apply (option_tac)
done

lemma mult_1_right_option [simp]:
fixes a :: "'a::monoid_mult option"
shows "a *? 1 = a"
apply (option_tac)
apply (induct_tac a; clarsimp)
done
```

3.6 ICL Interpretations

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

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

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

We make the above interpretation lemmas automatic simplifications.

```
declare preorder_equals_option.preorder_axioms [simp]
declare preorder_leq_option.preorder_axioms [simp]
declare preorder_subset_option.preorder_axioms [simp]
```

3.7 ICL Lifting Lemmas

```
lemma iclaw_eq_lift_option [simp]:
"iclaw (op =) seq_op par_op ⟹
 iclaw (op =?) seq_op↑? par_op↑?"
apply (unfold iclaw_def iclaw_axioms_def)
apply (option_tac)
done

lemma preorder_leq_lift_option [simp]:
"preorder (op ≤::'a::ord relop) ⟹
 preorder (op ≤?::'a::ord option relop)"
apply (unfold_locales)
apply (option_tac)
apply (meson preorder.refl)
apply (option_tac)
apply (meson preorder.trans)
done

lemma iclaw_leq_lift_option [simp]:
"iclaw (op ≤) seq_op par_op ⟹
 iclaw (op ≤?) seq_op↑? par_op↑?"
apply (unfold iclaw_def iclaw_axioms_def)
apply (option_tac)
done
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
```

We can easily prove that `number_range` is a non-empty set.

```
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

The above lemma enables us to introduce a 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(_)` will be used for the abstraction function.

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

The notation ⟦_⟧ will be used for the representation function.

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

setup_lifting type_definition_machine_number
```

4.3 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]
```

4.4 Instantiations

4.4.1 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

4.4.2 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
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
```

The notation ⊤ is introduced for the constructor `Overflow`.

```
adhoc_overloading global_top Overflow
```

5.2 Proof Support

Attribute used to collect definitional laws for operators.

```
named_theorems overflow_ops "definitional laws for operators on overflow values"
```

Tactic that facilitates proofs about `overflow` values.

```
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

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

5.3 Ordering Relation

Overflow ($\top$) 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
```

More instantiations can be added here as we desire.

```
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" where
[simp]: "overflow_return x = Value x"

adhoc_overloading return overflow_return
```

5.5 Generic Lifting

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"
```

We use the constant below for ad hoc overloading to avoid ambiguities.

```
consts lift_overflow :: "'a ⇒ 'b" ("_↑∞" [1000] 1000)

default_sort machine_number

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

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

default_sort type

adhoc_overloading lift_overflow ulift_overflow
adhoc_overloading lift_overflow blift_overflow
```

Note that we do not add the above operators to `overflow_ops`.

```
lemma ulift_overflow_simps [simp]:
"ulift_overflow f ⊤ = ⊤"
"ulift_overflow f (Value x) =
  (if (f ⟦x⟧) ≤ max_number then Value MN(f ⟦x⟧) else ⊤)"
apply (unfold ulift_overflow_def)
apply (simp_all)
done

lemma blift_overflow_simps [simp]:
"blift_overflow f x ⊤ = ⊤"
"blift_overflow f ⊤ y = ⊤"
"blift_overflow f (Value x') (Value y') =
  (if (f ⟦x'⟧ ⟦y'⟧) ≤ max_number then Value MN(f ⟦x'⟧ ⟦y'⟧) else ⊤)"
apply (unfold blift_overflow_def)
apply (simp_all)
apply (case_tac x; simp)
```

```
done
```

5.6 Lifted Operators

```
definition plus_overflow::
  "'a::{plus, machine_number} machine_number_ext binop" (infixl "+∞" 70) where
"plus_overflow = (op +)↑∞"

definition minus_overflow ::
  "'a::{minus, machine_number} machine_number_ext binop" (infixl "-∞" 70) where
"minus_overflow = (op -)↑∞"

definition times_overflow::
  "'a::{times, machine_number} machine_number_ext binop" (infixl "*∞" 70) where
"times_overflow = (op *)↑∞"

definition divide_overflow ::
  "'a::{divide, machine_number} machine_number_ext binop" (infixl "div∞" 70) where
"divide_overflow = (op div)↑∞"
```

Proof Support

```
declare plus_overflow_def [overflow_ops]
declare minus_overflow_def [overflow_ops]
declare times_overflow_def [overflow_ops]
declare divide_overflow_def [overflow_ops]
```

5.7 Instantiation Example

We give an instantiation for natural numbers.

```
instantiation nat :: machine_number
begin
definition max_number_nat :: "nat" where
"max_number_nat = 2 ^^ 32 - 1"
instance ..
end
```

5.8 Proof Experiments

```
lemma
fixes x :: "nat machine_number_ext"
fixes y :: "nat machine_number_ext"
shows "x *∞ y = y *∞ x"
— Is there another way to turn free variables in meta-quantified ones?
apply (transfer)
apply (overflow_tac)
apply (simp add: mult.commute)
done
```

Yes, using the below. Turn this into a tactic command! [TODO]

```
ML {* Induct.arbitrary_tac *}
end
```

6 Partiality

```
theory Partiality
imports Preliminaries ICL
begin
```

6.1 Type Definition

We define a datatype `'a partial` that adds a distinct ⊥ and ⊤ to a type `'a`.

```
datatype 'a partial =
  Bot | Value "'a" | Top
```

The notation ⊥ is introduced for the constructor `Bot`.

```
adhoc_overloading global_bot Bot
```

The notation ⊤ is introduced for the constructor `Top`.

```
adhoc_overloading global_top Top
```

6.2 Proof Support

Attribute used to collect definitional laws for operators.

```
named_theorems partial_ops "definitional laws for operators on partial values"
```

Tactic that facilitates proofs about `partial` values.

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

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

lemmas split_partial =
  split_partial_all
  split_partial_ex

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

6.3 Monadic Constructors

Note that we have to ensure strictness in both ⊥ and ⊤.

```
primrec partial_bind ::
  "'a partial ⇒ ('a ⇒ 'b partial) ⇒ 'b partial" where
```

```
"partial_bind Bot f = Bot" |
"partial_bind (Value x) f = f x" |
"partial_bind Top f = Top"
```

adhoc_overloading `bind partial_bind`

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

adhoc_overloading `return partial_return`

6.4 Generic Lifting

We use the constant below for ad hoc overloading to avoid ambiguities.

consts `lift_partial :: "'a ⇒ 'b" ("_↑`$_p$`" [1000] 1000)`

fun `ulift_partial :: "('a ⇒ 'b) ⇒ ('a partial ⇒ 'b partial)"` **where**
```
"ulift_partial f Bot = Bot" |
"ulift_partial f (Value x) = Value (f x)" |
"ulift_partial f Top = Top"
```

fun `blift_partial ::`
```
  "('a ⇒ 'b ⇒ 'c) ⇒ ('a partial ⇒ 'b partial ⇒ 'c partial)" where
"blift_partial f Bot Bot = Bot" |
"blift_partial f Bot (Value y) = Bot" |
"blift_partial f Bot Top = Bot" | — ⊥ dominates.
"blift_partial f (Value x) Bot = Bot" |
"blift_partial f (Value x) (Value y) = Value (f x y)" |
"blift_partial f (Value x) Top = Top" |
"blift_partial f Top Bot = Bot" | — ⊥ dominates.
"blift_partial f Top (Value y) = Top" |
"blift_partial f Top Top = Top"
```

adhoc_overloading `lift_partial ulift_partial`
adhoc_overloading `lift_partial blift_partial`

6.5 Lifted Operators

What about relational operators? How do we lift those? [TODO]

Addition and Subtraction

definition `plus_partial :: "'a::plus partial binop"` (**infixl** `"+`$_p$`" 70)` **where**
`"(op +`$_p$`) = (op +)↑`$_p$`"`

definition `minus_partial :: "'a::minus partial binop"` (**infixl** `"-`$_p$`" 70)` **where**
`"(op -`$_p$`) = (op -)↑`$_p$`"`

Multiplication and Division

definition `times_partial :: "'a::times partial binop"` (**infixl** `"*`$_p$`" 70)` **where**
`"(op *`$_p$`) = (op *)↑`$_p$`"`

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

Union and Disjoint Union

```
definition union_partial :: "'a set partial binop" (infixl "∪p" 70) where
"(op ∪p) = (op ∪)↑p"

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

Proof Support

```
declare plus_partial_def [partial_ops]
declare minus_partial_def [partial_ops]
declare times_partial_def [partial_ops]
declare divide_partial_def [partial_ops]
declare union_partial_def [partial_ops]
declare disjoint_union_def [partial_ops]
```

6.6 Ordering Relation

```
primrec partial_ord :: "'a partial ⇒ nat" where
"partial_ord Bot = 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
```

6.7 Class Instantiations

6.7.1 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
```

6.7.2 Partial Order

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

6.7.3 Linear Order

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

6.7.4 Lattice

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

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

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

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

Validation of the definition of meet and join above.

```
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
```

6.7.5 Complete Lattice

```
instantiation partial :: (complete_lattice) complete_lattice
begin
definition Inf_partial :: "'a partial set ⇒ 'a partial" where
[partial_ops]:
"Inf_partial xs =
  (if Bot ∈ xs then Bot 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 Bot 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
```

6.8 ICL Lifting Lemmas

```
lemma iclaw_eq_lift_partial [simp]:
"iclaw (op =) seq_op par_op ⟹
 iclaw (op =) seq_op↑p par_op↑p"
apply (unfold iclaw_def iclaw_axioms_def)
apply (partial_tac)
done

lemma preorder_less_eq_lift_partial [simp]:
"preorder (op ≤::'a::ord relop) ⟹
 preorder (op ≤::'a::ord partial relop)"
apply (unfold_locales)
apply (partial_tac)
apply (meson preorder.refl)
apply (partial_tac)
apply (meson preorder.trans)
done

lemma iclaw_less_eq_lift_partial [simp]:
"iclaw (op ≤) seq_op par_op ⟹
 iclaw (op ≤) seq_op↑p par_op↑p"
apply (unfold iclaw_def iclaw_axioms_def)
apply (partial_tac)
done
end
```

7 The Interchange Law

```
theory ICL
imports Preliminaries
begin
```

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

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

7.1 Locale Definitions

In this section, we encapsulate the interchange law via 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 us 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 the two operators. The interchange law thus becomes an assumption of the second locale.

7.1.1 Locale: `preorder`

The underlying relation has to be a preorder. Our definition of preorder is, however, deliberately weaker than Isabelle/HOL's, as encapsulated by its `ordering` locale. In particular, we shall not require the caveat `ordering ?less_eq ?less ⟹ ?less ?a ?b = (?less_eq ?a ?b ∧ ?a ≠ ?b)`. Moreover, interpretations only have to provide the ≤ operator and not `<` as well. We use bold-face symbols to distinguish our ordering relations from those of Isabelle's type classes.

```
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 ≤.

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

We can easily prove that ≡ is 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
end
```

7.1.2 Locale: iclaw

We next define the iclaw locale as an extension of the ICL.preorder locale above. The interchange law is encapsulated by the single assumption of the locale. Instantiations will have to discharge this assumption and thereby show that the interchange law holds for a particular type, ordering relation, and binary operator pair.

```
locale iclaw = preorder +
  fixes seq_op :: "'a binop" (infixr ";" 100)
  fixes par_op :: "'a binop" (infixr "|" 100)
  assumes interchange_law: "(p | r) ; (q | s) ≤ (p ; q) | (r ; s)"
```

7.2 Interpretations

We lastly prove a few useful interpretations of ICL.preorders. Due to the structuring mechanism of (sub)locales, we will later on be able to reuse these interpretation proofs when interpreting the iclaw locale for particular operators.

```
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_implies:
  preorder "TYPE(bool)" "op ⟶"
apply (unfold_locales)
apply (simp_all)
done

interpretation preorder_rimplies:
  preorder "TYPE(bool)" "op ⟵"
apply (unfold_locales)
apply (simp_all)
done
```

7.3 Proof Support

We make the above instantiation lemmas automatic simplifications.

```
declare preorder_eq.preorder_axioms [simp]
declare preorder_leq.preorder_axioms [simp]
declare preorder_implies.preorder_axioms [simp]
declare preorder_rimplies.preorder_axioms [simp]
end
```

8 Example Applications

```
theory ICL_Examples
imports ICL Strict_Operators Computer_Arith Partiality
begin

hide_const Partiality.Value
```

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

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

Example applications of the interchange law from the article.

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

We prove the interchange laws for the HOL types `int`, `rat` and `real`, as well as the corresponding `option` types of those. We note that the law does not hold for type `nat`, although a weaker version using ≤ instead of equality is provable because Isabelle/HOL interprets the minus operators as monus on natural numbers.

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

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

interpretation icl_plus_minus_nat_option:
  iclaw "TYPE(nat option)" "op ≤?" "op -?" "op +?"
apply (unfold_locales)
apply (option_tac)
done

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
```

Corresponding proofs for option types and strict operators.

```
interpretation icl_plus_minus_int_option:
  iclaw "TYPE(int option)" "op =?" "op -?" "op +?"
apply (unfold_locales)
apply (option_tac)
done

interpretation icl_plus_minus_rat_option:
  iclaw "TYPE(rat option)" "op =?" "op -?" "op +?"
apply (unfold_locales)
apply (option_tac)
done

interpretation icl_plus_minus_real_option:
  iclaw "TYPE(real option)" "op =?" "op -?" "op +?"
apply (unfold_locales)
apply (option_tac)
done
```

8.2 Positive arithmetic: with multiplication (×).

```
interpretation icl_plus_times_nat:
  iclaw "TYPE(nat)" "op ≤" "op +" "op *"
apply (unfold_locales)
apply (simp add: distrib_left distrib_right)
done

interpretation icl_plus_times_nat_option:
  iclaw "TYPE(nat option)" "op ≤?" "op +?" "op *?"
apply (unfold_locales)
apply (option_tac)
apply (simp add: distrib_left distrib_right)
done

interpretation icl_plus_times_nat_option:
  iclaw "TYPE(int)" "op ≤" "op +" "op *"
apply (unfold_locales)
apply (subgoal_tac "p ≥ 0 ∧ r ≥ 0 ∧ q ≥ 0 ∧ s ≥ 0")
— Subgoal 1
apply (clarify)
apply (unfold ring_distribs)
apply (unfold sym [OF add.assoc])
apply (simp)
oops
```

We note that the law can be proved more generally to hold in any (ordered) `semiring` in which `0::'a` is the least element. To mechanically verify this result, it is useful to introduce a type class that guarantees that all elements of a type are positive.

```
class positive = zero + ord +
  assumes zero_least: "0 ≤ x"

interpretation icl_positive_semiring:
  iclaw "TYPE('a::{positive,ordered_semiring})" "op ≤" "op +" "op *"
```

```
apply (unfold_locales)
apply (simp add: distrib_left distrib_right)
apply (metis add.right_neutral add_increasing add_mono order_refl zero_least)
done
```

Clearly, all elements of the type `nat` are positive.

```
instance nat :: positive
apply (intro_classes)
apply (simp)
done
```

For other number types, such as `integer`, `rational` and `real` numbers, we introduce a subtype `'a pos` that includes only the positive individuals of some type `'a`. In order to establish the non-emptiness caveat of the type definition, we require that the ordering be a `preorder`.

```
typedef (overloaded) 'a::"{zero, preorder}" pos = "{x::'a. 0 ≤ x}"
apply (clarsimp)
apply (rule_tac x = "0" in exI)
apply (rule order_refl)
done

setup_lifting type_definition_pos
```

We next lift '≤', '0', '+' and '*' into the new type `pos`.

```
instantiation pos :: ("{zero,preorder}") preorder
begin
lift_definition less_eq_pos :: "'a pos ⇒ 'a pos ⇒ bool"
is "op ≤" .
lift_definition less_pos :: "'a pos ⇒ 'a pos ⇒ bool"
is "op <" .
instance
apply (intro_classes; transfer)
using less_le_not_le apply (blast)
using order_refl apply (blast)
using order_trans apply (blast)
done
end

instantiation pos :: ("{zero,preorder}") zero
begin
lift_definition zero_pos :: "'a pos"
is "0" by (rule order_refl)
instance ..
end
```

We note that for the lifting of '+' and '*', we require closure of those operators under positive numbers. Such is, however, provable within ordered semi-rings, as we establish later on.

```
class plus_pos_cl = zero + ord + plus +
  assumes plus_pos_closure: "0 ≤ x ⟹ 0 ≤ y ⟹ 0 ≤ x + y"

class times_pos_cl = zero + ord + times +
  assumes times_pos_closure: "0 ≤ x ⟹ 0 ≤ y ⟹ 0 ≤ x * y"

instantiation pos :: ("{zero,preorder,plus_pos_cl}") plus
begin
```

```
lift_definition plus_pos :: "'a pos ⇒ 'a pos ⇒ 'a pos"
is "op +" by (rule plus_pos_closure)
instance ..
end

instantiation pos :: ("{zero,preorder,times_pos_cl}") times
begin
lift_definition times_pos :: "'a pos ⇒ 'a pos ⇒ 'a pos"
is "op *" by (rule times_pos_closure)
instance ..
end
```

We prove that the above closure property of '+' and '*' wrt the positive individuals holds within any (ordered) semi-ring.

```
subclass (in ordered_semiring) plus_pos_cl
apply (unfold class.plus_pos_cl_def)
using local.add_nonneg_nonneg by (blast)

subclass (in ordered_semiring_0) times_pos_cl
apply (unfold class.times_pos_cl_def)
using local.mult_nonneg_nonneg by (blast)
```

Lastly, we prove that subtype `'a pos` over some (ordered) semi-ring is itself and ordered semi-ring, albeit comprising positive elements only. With the earlier interpretation proof, namely for `icl_positive_semiring`, this implies that the interchange law holds for positive arithmetic with multiplication within any (ordered) semi-ring, including positive rational and real numbers.

```
instance pos :: ("{zero, preorder}") positive
apply (intro_classes)
apply (transfer)
apply (assumption)
done

instance pos :: (ordered_semiring_0) ordered_semiring
apply (intro_classes; transfer'; simp?)
apply (simp add: add.assoc)
apply (simp add: add.commute)
apply (simp add: add_left_mono)
apply (simp add: mult.assoc)
apply (simp add: distrib_right)
apply (simp add: distrib_left)
apply (simp add: mult_left_mono)
apply (simp add: mult_right_mono)
done

interpretation icl_plus_times_pos:
  iclaw "TYPE('a::ordered_semiring_0 pos)" "op ≤" "op +" "op *"
apply (unfold_locales)
done

interpretation icl_plus_times_pos_option:
  iclaw "TYPE('a::ordered_semiring_0 pos option)" "op ≤?" "op +?" "op *?"
apply (unfold_locales)
apply (option_tac)
apply (rule icl_positive_semiring.interchange_law)
done
```

```
interpretation icl_plus_times_pos_int:
  iclaw "TYPE(int pos)" "op ≤" "op +" "op *"
apply (unfold_locales)
done

interpretation icl_plus_times_pos_rat:
  iclaw "TYPE(rat pos)" "op ≤" "op +" "op *"
apply (unfold_locales)
done

interpretation icl_plus_times_pos_real:
  iclaw "TYPE(real pos)" "op ≤" "op +" "op *"
apply (unfold_locales)
done

interpretation icl_plus_times_pos_int_option:
  iclaw "TYPE(int pos option)" "op ≤?" "op +?" "op *?"
apply (unfold_locales)
done

interpretation icl_plus_times_pos_rat_option:
  iclaw "TYPE(rat pos option)" "op ≤?" "op +?" "op *?"
apply (unfold_locales)
done

interpretation icl_plus_times_pos_real_option:
  iclaw "TYPE(real pos option)" "op ≤?" "op +?" "op *?"
apply (unfold_locales)
done
```

8.3 Arithmetic: multiplication (×) and division (/) of numbers.

This is proved for 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

interpretation icl_mult_div_field:
  iclaw "TYPE('a::field)" "op =" "op *" "op /"
apply (unfold_locales)
apply (simp)
done

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

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

Theorem 1 likewise holds for `rational` and `real` numbers and option types thereof.

```
lemma Theorem1_rat:
fixes p :: "rat"
fixes q :: "rat"
shows "(p / q) * q = (p * q) / q"
apply (insert icl_mult_div_rat.interchange_law [of p q q 1])
apply (unfold div_by_1 mult_1_right)
apply (assumption)
done

lemma Theorem1_real:
fixes p :: "real"
fixes q :: "real"
shows "(p / q) * q = (p * q) / q"
apply (insert icl_mult_div_real.interchange_law [of p q q 1])
apply (unfold div_by_1 mult_1_right)
apply (assumption)
done

lemma Theorem1_rat_option:
fixes p :: "rat option"
fixes q :: "rat option"
shows "(p /? q) *? q = (p *? q) /? q"
apply (insert icl_mult_div_rat_option.interchange_law [of p q q 1])
apply (unfold div_by_1_option mult_1_right_option)
apply (case_tac p; case_tac q; option_tac)
done

lemma Theorem1_real_option:
fixes p :: "real option"
fixes q :: "real option"
shows "(p /? q) *? q = (p *? q) /? q"
apply (insert icl_mult_div_real_option.interchange_law [of p q q 1])
apply (unfold div_by_1_option mult_1_right_option)
apply (case_tac p; case_tac q; option_tac)
done
```

It also holds, more generally, in any division ring.

```
context division_ring
begin
lemma div_mult_exchange:
fixes p :: "'a"
fixes q :: "'a"
shows "(p / q) * q = (p * q) / q"
apply (metis eq_divide_eq mult_eq_0_iff)
done
```

```
end
```

8.4 Positive integers: with truncated division (÷).

By default, `x div y` is also used for truncated (integer) division in Isabelle/HOL. Hence, we first introduce a neat syntax `x ÷ y` consistent with our notation in the paper. This is done via **abbreviation**.

```
abbreviation trunc_div :: "nat binop" (infixl "÷" 70) where
"x ÷ y ≡ x div y"

abbreviation trunc_div_option :: "nat option binop" (infixl "÷?" 70) where
"x ÷? y ≡ x /? y"
```

Since Isabelle/HOL defines `x div 0 = 0`, we can prove the interchange law even in HOL's weak treatment of undefinedness, as well as in the strong one.

```
interpretation icl_mult_trunc_div_nat:
  iclaw "TYPE(nat)" "op ≤" "op *" "op ÷"
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 mult.commute mult_le_mono split_div_lemma)
done

interpretation icl_mult_trunc_div_nat_option:
  iclaw "TYPE(nat option)" "op ≤?" "op *?" "op ÷?"
apply (unfold_locales)
apply (option_tac)
apply (rule icl_mult_trunc_div_nat.interchange_law)
done
```

With the above, we prove Theorem 2 in the paper, both for natural numbers and the option type over naturals.

```
lemma Theorem2:
fixes p :: "nat"
fixes q :: "nat"
shows "(p ÷ q) * q ≤ (p * q) ÷ q"
apply (insert icl_mult_trunc_div_nat.interchange_law [of p q q 1])
apply (unfold div_by_1 mult_1_right)
apply (assumption)
done

lemma Theorem2_option:
fixes p :: "nat option"
fixes q :: "nat option"
shows "(p ÷? q) *? q ≤ (p *? q) ÷? q"
apply (insert icl_mult_trunc_div_nat_option.interchange_law [of p q q 1])
apply (unfold div_by_1_option mult_1_right_option)
apply (case_tac p; case_tac q; option_tac)
done
```

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

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

We can easily verify the above equivalence in HOL.

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

This instance of the interchange law cannot be proved by way of interpreting the `iclaw` locale because rule implication is not an object-logic operator. Nonetheless, we can prove the interchange law as an Isabelle/HOL proof rule. We note that Isabelle uses ⟶ for implication and ⟹ for meta-level (rule) implication. To make the theorem look as in the paper, we temporarily change the syntax of those operators.

```
notation HOL.implies (infixr "⇒" 25)
notation Pure.imp    (infixr "⊢" 1)

lemma icl_conj_imp_prop:
"(p ⇒ q) ∧ (r ⇒ s) ⊢ (p ∧ r) ⇒ (q ∧ s)"
apply (auto)
done
```

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

Numerical value of a `boolean`.

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

Order on boolean values induced by `valOfBool`.

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

We show that the numerical order above 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

interpretation preorder_numOrdBool:
  preorder "TYPE(bool)" "numOrdBool"
apply (unfold_locales)
apply (unfold numOrdBool_is_imp)
apply (auto)
done
```

Note that `;` is ∨ and `|` is ∧.

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

```
done
```

Theorem 3 once again needs to be formulated as an Isabelle proof rule.

```
lemma Theorem3:
"q ∧ s ⊢ q ∨ s"
apply (auto)
done

no_notation HOL.implies (infixr "⇒" 25)
no_notation Pure.imp    (infixr "⊢" 1)
```

8.7 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 (o) to avoid clashes with Isabelle/HOL's symbol (∘) for functional composition.

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

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

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

We first show that any associative and commuting operator self-interchanges.

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

We next show that self-interchanging operators with a unit are associative and commute (Theorem 4).

```
lemma Theorem4_assoc:
"(self_iclaw bop) ∧ (has_unit bop one) ⟹ associative bop"
apply (unfold_locales)
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 Theorem4_commute:
"(self_iclaw bop) ∧ (has_unit bop one) ⟹ commutative bop"
apply (unfold_locales)
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 +, *, ∨ and ∧.

```
interpretation self_icl_plus:
  self_iclaw "TYPE('a::comm_monoid_add)" "op +"
apply (rule assoc_comm_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_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_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_self_iclaw)
apply (rule conjI)
— Subgoal 1
apply (standard) [1]
apply (blast)
— Subgoal 2
apply (standard) [1]
apply (blast)
done
```

In addition, we can also show self-interchanging of +? and *?.

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

interpretation self_icl_mult_option:
  self_iclaw "TYPE('a::comm_monoid_mult option)" "op *?"
apply (rule assoc_comm_self_iclaw)
apply (rule conjI)
— Subgoal 1
apply (unfold associative_def)
apply (option_tac)
apply (simp add: mult.assoc)
— Subgoal 2
apply (unfold commutative_def)
apply (option_tac)
apply (simp add: mult.commute)
done
```

8.8 Note: Partial operators.

TO: This validates the cancellation law in the algebra of Section 4.

Note that the below could even be proved if removing the assumption `0 < q`. The reason for this is that in Isabelle/HOL, division by zero is defined to be zero. Below we, however, conduct the prove not exploiting that fact.

```
lemma trunc_div_mult_cancel:
fixes p :: "nat"
fixes q :: "nat"
assumes "0 < q"
shows "(p ÷ q) * q ≤ p"
apply (insert Theorem2 [of p q])
apply (erule order_trans)
```

apply (simp)
done

lemma trunc_div_mult_cancel_option:
fixes p :: "nat option"
fixes q :: "nat option"
shows "(p $\div_?$ q) $*_?$ q $\leq$ p"
apply (induction p; induction q; option_tac)
apply (rename_tac q p)
apply (erule trunc_div_mult_cancel)
done

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

We note that the various necessary types and operators to formalise machine calculations are developed in the theories:

- Strict_Operators;
- Machine_Number;
- Overflow_Monad; and
- Computer_Arith.

Cancellation Laws

lemma Section_8_cancel_law_1a:
fixes p :: "nat machine_number_ext"
fixes q :: "nat machine_number_ext"
shows "q $\neq$ 0 $\Longrightarrow$ p $\leq$ (p $*_\infty$ q) div$_\infty$ q"
apply (transfer) — Just to quantify free variables!
apply (overflow_tac)
done

lemma Section_8_cancel_law_1b:
fixes p :: "nat comparith"
fixes q :: "nat comparith"
shows "q $\neq$ 0 $\Longrightarrow$ q $\neq$ $\perp$ $\Longrightarrow$ p $\leq$ (p $*_c$ q) $/_c$ q"
apply (transfer) — Just to quantify free variables!
apply (comparith_tac)
done

lemma Section_8_cancel_law_2a:
fixes p :: "nat option"
fixes q :: "nat option"
shows "(p $/_?$ q) $*_?$ q $\leq$ p"
apply (transfer) — Just to quantify free variables!
apply (option_tac)
apply (metis mult.commute split_div_lemma)
done

lemma Section_8_cancel_law_2b:
fixes p :: "nat comparith"
fixes q :: "nat comparith"

shows "q $\neq$ $\top$ $\Longrightarrow$ (p $/_c$ q) $*_c$ q $\leq$ p"
apply (transfer) — Just to quantify free variables!
apply (comparith_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 (metis dual_order.trans mult.commute split_div_lemma)
done

Interchange Law

lemma overflow_times_neq_Value_MN_0:
fixes x :: "nat machine_number_ext"
fixes y :: "nat machine_number_ext"
shows
"x $\neq$ Value MN(0) $\Longrightarrow$
 y $\neq$ Value MN(0) $\Longrightarrow$ x $*_\infty$ y $\neq$ Value MN(0)"
apply (transfer) — Just to quantify free variables!
apply (overflow_tac)
done

interpretation icl_mult_trunc_div_nat_overflow:
  iclaw "TYPE(nat comparith)" "op $\leq$" "op $*_c$" "op $/_c$"
apply (unfold_locales)
apply (option_tac)
apply (simp add: overflow_times_neq_Value_MN_0)
apply (unfold times_overflow_def divide_overflow_def)
apply (thin_tac "r $\neq$ Value MN(0)")
apply (thin_tac "s $\neq$ 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

8.10 Sets: union ($\cup$) and disjoint union (+) of sets, ordered by inclusion $\subseteq$.

Proof of the below relies on $\perp$ $\subseteq_?$ A for any A.

interpretation preorder_option_subset:
  iclaw "TYPE('a set option)" "(op $\subseteq_?$)" "op $\oplus_?$" "op $\cup_?$"
apply (unfold_locales)
apply (rename_tac p q r s)
apply (option_tac)
apply (auto)
oops

RE: Disjoint union has a unit {}, and so it interchanges with itself.

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

interpretation self_icl_disjoint_union:
  self_iclaw "TYPE('a set option)" "op ⊕?"
apply (unfold_locales)
apply (option_tac)
apply (auto)
done
```

RE: But it is clearly not idempotent: `p ⊕? p = p` only when `p = {}` or `p = ⊥` or `p = ⊤`

TODO: Use the type `partial` to prove this also for ⊤.

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

8.11 Note: Variance of operators, covariant (+,∧, ∨) and contravariant (-, ∧, ⇐)

We introduce the property of covariance and contravariance via locales. For covariance, we have a single locale; and for contravariance, three different locales to account for all possible combinations.

```
locale covariant = preorder +
  fixes cov_op :: "'a binop" (infixr "cov" 100)
  assumes cov_rule: "x ≤ x' ∧ y ≤ y' ⟹ (x cov y) ≤ (x' cov y')"
```

We consider contravariance in the first, second or both operators.

```
locale contravariant = preorder +
  fixes cot_op :: "'a binop" (infixr "cot" 100)
  assumes cot_rule: "x' ≤ x ∧ y' ≤ y ⟹ (x cot y) ≤ (x' cot y')"

locale contravariant1 = preorder +
  fixes cot_op :: "'a binop" (infixr "cot" 100)
  assumes cot_rule1: "x' ≤ x ∧ y ≤ y' ⟹ (x cot y) ≤ (x' cot y')"

locale contravariant2 = preorder +
  fixes cot_op :: "'a binop" (infixr "cot" 100)
  assumes cot_rule2: "x ≤ x' ∧ y' ≤ y ⟹ (x cot y) ≤ (x' cot y')"
```

Note that if the ordering is equality, all operators are covariant.

```
interpretation covariant_equality:
  covariant "TYPE('a)" "op =" "f::'a binop"
apply (intro_locales)
apply (unfold covariant_axioms_def)
apply (clarsimp)
done

interpretation contravariant_equality:
  contravariant "TYPE('a)" "op =" "f::'a binop"
apply (intro_locales)
apply (unfold contravariant_axioms_def)
apply (clarsimp)
done

interpretation contravariant1_equality:
  contravariant1 "TYPE('a)" "op =" "f::'a binop"
apply (intro_locales)
apply (unfold contravariant1_axioms_def)
apply (clarsimp)
done

interpretation contravariant2_equality:
  contravariant2 "TYPE('a)" "op =" "f::'a binop"
apply (intro_locales)
apply (unfold contravariant2_axioms_def)
apply (clarsimp)
done
```

Below, we prove covariance of `+` for `natural`, `integer`, `rational` and `real` numbers, as well as extensions of those types with ⊥.

```
interpretation covariant_plus_nat:
  covariant "TYPE(nat)" "op ≤" "op +"
apply (unfold_locales)
apply (linarith)
done

interpretation covariant_plus_int:
  covariant "TYPE(int)" "op ≤" "op +"
apply (unfold_locales)
apply (linarith)
done

interpretation covariant_plus_rat:
  covariant "TYPE(rat)" "op ≤" "op +"
apply (unfold_locales)
apply (linarith)
done

interpretation covariant_plus_real:
  covariant "TYPE(real)" "op ≤" "op +"
apply (unfold_locales)
apply (linarith)
done

interpretation covariant_plus_nat_option:
  covariant "TYPE(nat option)" "op ≤?" "op +?"
apply (unfold_locales)
apply (option_tac)
done

interpretation covariant_plus_int_option:
  covariant "TYPE(int option)" "op ≤?" "op +?"
apply (unfold_locales)
apply (option_tac)
done

interpretation covariant_plus_rat_option:
  covariant "TYPE(rat option)" "op ≤?" "op +?"
apply (unfold_locales)
apply (option_tac)
done

interpretation covariant_plus_real_option:
  covariant "TYPE(real option)" "op ≤?" "op +?"
apply (unfold_locales)
apply (option_tac)
done
```

Covariance of conjunction and disjunction with respect to implication.

```
interpretation covariant_conj:
  covariant "TYPE(bool)" "op ⟶" "op ∧"
apply (unfold_locales)
apply (clarsimp)
done

interpretation covariant_disj:
  covariant "TYPE(bool)" "op ⟶" "op ∨"
apply (unfold_locales)
apply (clarsimp)
done
```

We prove contravariance in the right operator of `-` for `nat`ural, `int`eger, `rat`ional and `real` numbers. We note that contravariance does not hold for their respective `option` types. A counter examples is where `y' = ⊥` in `(x cov y) ≤ (x' cov y')` with all other quantities defined.

```
interpretation contravariant2_minus_nat:
  contravariant2 "TYPE(nat)" "op ≤" "op -"
apply (unfold_locales)
apply (linarith)
done

interpretation contravariant2_minus_int:
  contravariant2 "TYPE(int)" "op ≤" "op -"
apply (unfold_locales)
apply (linarith)
done

interpretation contravariant2_minus_rat:
  contravariant2 "TYPE(rat)" "op ≤" "op -"
apply (unfold_locales)
apply (linarith)
done

interpretation contravariant2_minus_real:
  contravariant2 "TYPE(real)" "op ≤" "op -"
apply (unfold_locales)
apply (linarith)
done
```

Contravariance of division actually could not be proved. First of all it does not hold for plain number types `nat` since the additional caveat `(0::'a) < y'` is needed, see the proof below. For `int`, `rat` and `real` it is even worse, since we also need to show that `y*y'` is positive. Moving to `option` types does not help as we are facing the same issue as for `-` above. Various instances of the contravariance law for division may only be proved if we strengthen the assumptions on `y` and `y'`.

```
interpretation contravariant2_nat:
  contravariant2 "TYPE(nat)" "op ≤" "op div"
apply (unfold_locales)
apply (clarify)
apply (subgoal_tac "x div y ≤ x div y'")
apply (erule order_trans)
apply (erule div_le_mono)
apply (rule div_le_mono2)
apply (simp_all)
oops

interpretation contravariant2_rat:
  contravariant2 "TYPE(rat)" "op ≤" "op /"
apply (unfold_locales)
apply (clarify)
apply (subgoal_tac "x / y ≤ x / y'")
apply (erule order_trans)
apply (erule divide_right_mono) defer
apply (erule divide_left_mono) defer
defer
oops

interpretation contravariant2_div_nat:
  contravariant2 "TYPE(nat option)" "op ≤?" "op /?"
apply (unfold_locales)
apply (option_tac)
apply (safe; clarsimp?) defer
apply (subgoal_tac "x div y ≤ x div y'")
apply (erule order_trans)
apply (erule div_le_mono)
apply (erule div_le_mono2)
apply (assumption)
oops
```

Contravariance in the second operators holds for reverse implication.

```
interpretation contravariant_ref_implies:
  contravariant2 "TYPE(bool)" "op ⟶" "op ⟵"
apply (unfold_locales)
apply (auto)
done
```

Covariance and contravariance with respect to equality is trivial in HOL due to Leibniz's law following from the axioms of the HOL kernel.

8.12 Note: Modularity, compositionality, locality, etc.

This proof could be more involved in requiring inductive reasoning about arbitrary languages whose operators are covariant with respect to an order. In a deep embedding of a specific language, this would not be difficult to show. We will not dig deeper into mechanically proving this property in all its generality, as it requires deep embedding of HOL functions, and giving a semantics to this (in HOL) I stipulate is beyond expressivity of the type system of HOL. An inductive proof would have to proceed at the meta-level.

8.13 Strings of characters: catenation (;) interleaving (|) and empty string (ε).

We first define a datatype to formalise the syntax of our string algebra.

Note that we added a constructor for a single character (`atom`).

```
datatype 'a str_calc =
  empty_str ("ε") |
  atom "'a" |
  seq_str "'a str_calc" "'a str_calc" (infixr ";" 110) |
  par_str "'a str_calc" "'a str_calc" (infixr "|" 100)
```

The following function facilitates construction from HOL strings.

```
primrec mk_str :: "string ⇒ char str_calc"  where
"mk_str [] = ε" |
"mk_str (h # t) = seq_str (atom h) (mk_str t)"

syntax "_mk_str" :: "id ⇒ char str_calc" ("≪_≫")

parse_translation ‹
  let
    fun mk_str_tr [Free  (name, _)] = @{const mk_str} $ (HOLogic.mk_string name)
      | mk_str_tr [Const (name, _)] = @{const mk_str} $ (HOLogic.mk_string name)
      | mk_str_tr _ = raise Match;
  in
    [(@{syntax_const "_mk_str"}, K mk_str_tr)]
  end
›

translations "_mk_str s" ↽ "(CONST mk_str) s"
```

The function `ch` yields all characters in a `str_calc` term.

```
primrec ch :: "'a str_calc ⇒ 'a set" where
"ch ε = {}" |
"ch (atom c) = {c}" |
"ch (p ; q) = (ch p) ∪ (ch q)" |
"ch (p | q) = (ch p) ∪ (ch q)"
```

The function `sd` computes the sequential dependencies using `ch`.

```
primrec sd :: "'a str_calc ⇒ ('a × 'a) set" where
"sd ε = {}" |
"sd (atom c) = {}" |
"sd (p ; q) = {(c, d). c ∈ (ch p) ∧ d ∈ (ch q)} ∪ sd(p) ∪ sd(q)" |
"sd (p | q) = sd(p) ∪ sd(q)"
```

We are now able to define our ordering of `str_calc` objects.

```
instantiation str_calc :: (type) ord
begin
```

```
definition less_eq_str_calc :: "'a str_calc ⇒ 'a str_calc ⇒ bool" where
"less_eq_str_calc p q ⟷ (*ch p = ch q ∧*)sd(q) ⊆ sd(p)"
definition less_str_calc :: "'a str_calc ⇒ 'a str_calc ⇒ bool" where
"less_str_calc p q ⟷ (*ch p = ch q ∧*)sd(q) ⊂ sd(p)"
instance ..
end
```

Proof of the interchange law for the string calculus operators.

```
instance str_calc :: (type) preorder
apply (intro_classes)
apply (unfold less_eq_str_calc_def less_str_calc_def)
apply (auto)
done

interpretation preorder_str_calc:
  preorder "TYPE('a str_calc)" "op ≤"
apply (rule ICL.preorder_leq.preorder_axioms)
done

interpretation iclaw_str_calc:
  iclaw "TYPE('a str_calc)" "op ≤" "op ;" "op |"
apply (unfold_locales)
apply (unfold less_eq_str_calc_def less_str_calc_def)
apply (clarsimp)
apply (simp add: subset_iff)

done
```

8.14 Note: Small interchange laws.

```
lemma equiv_str_calc:
"s ≅ t ⟷ (*ch s = ch t ∧*) sd s = sd t"
apply (clarsimp)
apply (unfold less_eq_str_calc_def)
apply (auto)
done

lemma empty_str_seq_unit:
"ε ; s ≅ s"
"s ; ε ≅ s"
apply (unfold equiv_str_calc)
apply (auto)
done

lemma empty_str_par_unit:
"ε | s ≅ s"
"s | ε ≅ s"
apply (unfold equiv_str_calc)
apply (auto)
done

lemma small_interchange_laws:
"(p | q) ; s ≤ p | (q ; s)"
"p ; (r | s) ≤ (p ; r) | s"
"q ; (r | s) ≤ r | (q ; s)"
"(p | q) ; r ≤ (p ; r) | q"
```

```
"p ; s ≤ p | s"
"q ; s ≤ s | q"
apply (unfold less_eq_str_calc_def)
apply (auto)
done
```

8.15 Note: an example derivation

We first prove several key lemmas.

```
lemma seq_str_assoc:
"(s ; t) ; u ≥ s ; t ; u"
apply (unfold less_eq_str_calc_def)
apply (auto)
done

lemma par_str_assoc:
"(s | t) | u ≥ s | t | u"
apply (unfold less_eq_str_calc_def)
apply (auto)
done
```

The following law does not hold but is needed to remove the `ch`-related provisos in the law `seq_str_mono`. Alternatively, we could strengthen the definition of the order by additionally requiring `ch p = ch q`.

```
lemma sd_imp_ch_subset:
"sd s ⊆ sd t ⟹ ch s ⊆ ch t"
apply (induction s; induction t)
apply (simp)
apply (simp)
apply (simp)
apply (simp)
defer
defer
apply (simp)
apply (simp)
apply (simp)
apply (simp)
apply (simp)
apply (simp)
apply (simp)
apply (simp)
apply (simp)
apply (simp)
oops

lemma seq_str_mono:
"ch s = ch s' ⟹
 ch t = ch t' ⟹
 s ≥ s' ⟹ t ≥ t' ⟹ (s ; t) ≥ (s' ; t')"
apply (unfold less_eq_str_calc_def)
apply (auto)
done

lemma par_str_mono:
"s ≥ s' ⟹ t ≥ t' ⟹ (s | t) ≥ (s' | t')"
```

```
apply (unfold less_eq_str_calc_def)
apply (auto)
done

lemma str_calc_step:
fixes LHS :: "'a::preorder"
fixes RHS :: "'a::preorder"
fixes MID :: "'a::preorder"
shows "LHS ≥ MID ⟹ MID ≥ RHS ⟹ LHS ≥ RHS"
using order_trans by (blast)

lemma example_derivation:
assumes lhs: "LHS = «abcd» | «xyzw»"
assumes rhs: "RHS = «xaybzwcd»"
shows "LHS ≥ RHS"
apply (unfold lhs rhs)
— Step 1
apply (rule_tac MID = "(«a» ; «bcd») | («xy» ; «zw»)" in str_calc_step)
apply (unfold less_eq_str_calc_def; auto) [1]
— Step 2
apply (rule_tac MID = "(«a» | «xy») ; («bcd» | «zw»)" in str_calc_step)
apply (rule iclaw_str_calc.interchange_law)
— Step 3
apply (rule_tac MID = "(«a» | «x» ; «y») ; («b» ; «cd» | «zw»)" in str_calc_step)
apply (unfold less_eq_str_calc_def; auto) [1]
— Step 4
apply (rule_tac MID = "(«a» | «x») ; «y» ; («b» | «zw») ; «cd»" in str_calc_step)
apply (unfold less_eq_str_calc_def; auto) [1]

— Remainder of the proof...
apply (unfold less_eq_str_calc_def)
apply (auto)
done

lemma example_derivation_auto:
assumes lhs: "LHS = «abcd» | «xyzw»"
assumes rhs: "RHS = «xaybzwcd»"
shows "LHS ≥ RHS"
apply (unfold lhs rhs)
apply (unfold less_eq_str_calc_def)
apply (auto)
done
end
```