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

Tony Hoare   Bernard Möller   Georg Struth   Frank Zeyda

July 23, 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 Monad Syntax

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

consts return :: "'a ⇒ 'b" ("return")

1.4 Type Class for Equivalence

class equiv = ord +
  fixes equiv :: "'a ⇒ 'a ⇒ bool" (**infix** "≅" 50)
  assumes equiv_mutual_ord : "x ≅ y ⟷ x ≤ y ∧ y ≤ x"

lemma equiv_relf:
fixes x :: "'a::{equiv, preorder}"
shows "x ≅ x"
apply (unfold equiv_mutual_ord)
apply (clarsimp)
done

lemma equiv_sym:

```
"x ≅ y ⟹ y ≅ x"
apply (unfold equiv_mutual_ord)
apply (auto)
done

lemma equiv_trans:
fixes x :: "'a::{equiv, preorder}"
fixes y :: "'a::{equiv, preorder}"
fixes z :: "'a::{equiv, preorder}"
shows "x ≅ y ⟹ y ≅ z ⟹ x ≅ z"
apply (unfold equiv_mutual_ord)
using order_trans by (auto)
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 $\bot$ 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` $\Rightarrow$ `'a option"` **where**
`[simp]: "option_return x = Some x"`

adhoc_overloading `return option_return`

Note that `op` $\ggg$ 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 $\leq$ 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 ⟷ (*True*) False"` |
`"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 $*$)$\uparrow_?$"

definition `divide_option` :: "'a::{divide, zero} option binop" (**infixl** "'/$_?$" 70) **where**
"x /$_?$ y = do {x' $\leftarrow$ x; y' $\leftarrow$ y; if y' $\neq$ 0 then return (x' div y') else $\bot$}"

Union and Disjoint Union

definition `union_option` :: "'a set option binop" (**infixl** "$\cup_?$" 70) **where**
"(op $\cup_?$) = (op $\cup$)$\uparrow_?$"

definition `disjoint_union` :: "'a set option binop" (**infixl** "$\oplus_?$" 70) **where**
"x $\oplus_?$ y = do {x' $\leftarrow$ x; y' $\leftarrow$ y; if x' $\cap$ y' = {} then return (x' $\cup$ y') else $\bot$}"

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   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 $\leq_?$)"
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 $\subseteq_?$)"
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]

3.6   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 $\top$.

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

The notation $\top$ 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** `"+`$_\infty$`"` **70**) **where**
`"plus_overflow = (op +)`$\uparrow_\infty$`"`

definition `minus_overflow ::`
  `"'a::{minus, machine_number} machine_number_ext binop"` (**infixl** `"-`$_\infty$`"` **70**) **where**
`"minus_overflow = (op -)`$\uparrow_\infty$`"`

definition `times_overflow::`
  `"'a::{times, machine_number} machine_number_ext binop"` (**infixl** `"*`$_\infty$`"` **70**) **where**
`"times_overflow = (op *)`$\uparrow_\infty$`"`

definition `divide_overflow ::`
  `"'a::{divide, machine_number} machine_number_ext binop"` (**infixl** `"div`$_\infty$`"` **70**) **where**
`"divide_overflow = (op div)`$\uparrow_\infty$`"`

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 *`$_\infty$` y = y *`$_\infty$` 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 $\bot$ and $\top$ to a type `'a`.

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

The notation $\bot$ is introduced for the constructor `Bot`.

adhoc_overloading `global_bot Bot`

The notation $\top$ 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:`
`"(`$\forall$`x::'a partial. P x) = (P Bot `$\wedge$` P Top `$\wedge$` (`$\forall$`x::'a. P (Value x)))"`
apply `(safe; simp?)`
apply `(case_tac x)`
apply `(simp_all)`
done

lemma `split_partial_ex:`
`"(`$\exists$`x::'a partial. P x) = (P Bot `$\vee$` P Top `$\vee$` (`$\exists$`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 $\bot$ and $\top$.

primrec `partial_bind ::`
  `"'a partial `$\Rightarrow$` ('a `$\Rightarrow$` 'b partial) `$\Rightarrow$` '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** "$\cup_p$" 70) **where**
"(op $\cup_p$) = (op $\cup$)$\uparrow_p$"

definition `disjoint_union` :: "'a set partial binop" (**infixl** "$\oplus_p$" 70) **where**
"x $\oplus_p$ y = do {x' $\leftarrow$ x; y' $\leftarrow$ y; if x' $\cap$ y' = {} then return (x' $\cup$ y') else $\bot$}"

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 $\Rightarrow$ 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 $\Rightarrow$ 'a partial $\Rightarrow$ bool" **where**
"(Value x) $\leq$ (Value y) $\longleftrightarrow$ x $\leq$ y" |
"a $\leq$ b $\longleftrightarrow$ (partial_ord a) $\leq$ (partial_ord b)"

fun `less_partial` :: "'a partial $\Rightarrow$ 'a partial $\Rightarrow$ bool" **where**
"(Value x) < (Value y) $\longleftrightarrow$ x < y" |
"a < b $\longleftrightarrow$ (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 $\Rightarrow$ 'a partial $\Rightarrow$ 'a partial" **where**
"Bot $\sqcap$ Bot = Bot" |
"Bot $\sqcap$ (Value y) = Bot" |
"Bot $\sqcap$ Top = Bot" |
"(Value x) $\sqcap$ Bot = Bot" |
"(Value x) $\sqcap$ (Value y) = Value (x $\sqcap$ y)" |
"(Value x) $\sqcap$ Top = (Value x)" |
"Top $\sqcap$ Bot = Bot" |
"Top $\sqcap$ Value y = Value y" |
"Top $\sqcap$ Top = Top"

fun sup_partial :: "'a partial $\Rightarrow$ 'a partial $\Rightarrow$ 'a partial" **where**
"Bot $\sqcup$ Bot = Bot" |
"Bot $\sqcup$ (Value y) = (Value y)" |
"Bot $\sqcup$ Top = Top" |
"(Value x) $\sqcup$ Bot = (Value x)" |
"(Value x) $\sqcup$ (Value y) = Value (x $\sqcup$ y)" |
"(Value x) $\sqcup$ Top = Top" |
"Top $\sqcup$ Bot = Top" |
"Top $\sqcup$ (Value y) = Top" |
"Top $\sqcup$ 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]:
"$\forall$a b. partial_ord (a $\sqcap$ b) = min (partial_ord a) (partial_ord b)"
apply (partial_tac)
done

lemma partial_ord_sup_lemma [simp]:
"$\forall$a b. partial_ord (a $\sqcup$ 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 $\Rightarrow$ 'a partial" **where**
[partial_ops]:
"Inf_partial xs =
  (if Bot $\in$ xs then Bot else
    let values = {x. Value x $\in$ xs} in
      if values = {} then Top else Value (Inf values))"

definition Sup_partial :: "'a partial set $\Rightarrow$ 'a partial" **where**
[partial_ops]:
"Sup_partial xs =
  (if Top $\in$ xs then Top else
    let values = {x. Value x $\in$ 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` $\implies$
 `iclaw (op =) seq_op`$\uparrow_p$ `par_op`$\uparrow_p$`"`
apply `(unfold iclaw_def iclaw_axioms_def)`
apply `(partial_tac)`
done

lemma `preorder_less_eq_lift_partial` `[simp]`:
`"preorder (op` $\leq$`::'a::ord relop)` $\implies$
 `preorder (op` $\leq$`::'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` $\leq$`) seq_op par_op` $\implies$
 `iclaw (op` $\leq$`) seq_op`$\uparrow_p$ `par_op`$\uparrow_p$`"`
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` $\implies$ `?less ?a ?b = (?less_eq ?a ?b` $\land$ `?a` $\neq$ `?b)`. Moreover, interpretations only have to provide the $\leq$ 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 $\Rightarrow$ 'a $\Rightarrow$ bool" (**infix** "$\leq$" 50)
  assumes refl: "x $\leq$ x"
  assumes trans: "x $\leq$ y $\implies$ y $\leq$ z $\implies$ x $\leq$ z"
begin

Equivalence $\equiv$ of elements is defined in terms of mutual $\leq$.

definition equiv :: "'a $\Rightarrow$ 'a $\Rightarrow$ bool" (**infix** "$\equiv$" 50) **where**
"x $\equiv$ y $\longleftrightarrow$ x $\leq$ y $\land$ y $\leq$ x"

We can easily prove that $\equiv$ is an equivalence relation.

lemma equiv_refl:
"x $\equiv$ x"
apply (unfold equiv_def)
apply (clarsimp)
apply (rule local.refl)
done

lemma equiv_sym:
"x $\equiv$ y $\implies$ y $\equiv$ x"
apply (unfold equiv_def)
apply (clarsimp)
done

lemma equiv_trans:
"x $\equiv$ y $\implies$ y $\equiv$ z $\implies$ x $\equiv$ 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
```

7.3 Proof Support

We make the above instantiation lemmas automatic simplifications.

```
declare preorder_eq.preorder_axioms [simp]
declare preorder_leq.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 working note.

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

We prove the ICL for the HOL types `int`, `rat` and `real`, as well as the corresponding `option` type instances of those types.

— 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

Corresponding ICL 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   Arithmetic: multiplication (x) 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_rat_option`:
  iclaw "TYPE(rat option)" "op =?" "op *?" "op /?"
apply (unfold_locales)
apply (option_tac)
done

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

The theorem below holds for 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

... and hence for `rational` and `real` numbers.

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

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

8.3 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 nat_div_mult_leq:
fixes p :: "nat"
fixes q :: "nat"
```
— The assumption `q > 0` is not needed because of `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
```

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 our strong 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 nat_div_mult_leq)
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
```

8.4 Propositional calculus: conjunction ($\wedge$) and implication ($\Rightarrow$).

RE: Implication $p \Rightarrow q$ is defined in the usual way as $\neg p \vee q$.

We can easily verify the above equivalence in HOL.

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

8.5 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

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

8.6 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 (⚬) to avoid clashes with 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"
```

Finally, we can set to prove the laws in the paper.

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

```
lemma self_iclaw_and_unit_imp_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 self_iclaw_and_unit_imp_comm:
"(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_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 $\wedge$"
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 $\vee$"
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

8.7 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 $\implies$ 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 $\implies$ q $\neq$ $\bot$ $\implies$ 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 $\neq$ $\top$ $\implies$ (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) $\implies$
 y $\neq$ Value MN(0) $\implies$ 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
```

8.8   Note: Partial operators.

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

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

Talk to Tony and Georg about the failed proof below! [TODO]

```
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

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 = ⊤`

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

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

8.10   Strings of characters: catenation (;) interleaving ∣ and empty string (e).

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

Note that I 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" (infix ";" 100) |
  par_str "'a str_calc" "'a str_calc" (infix "|" 100)
```

The following function facilitates the construction of strings.

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

value "≪''frank''≫"

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 the sequential dependencies and uses ch for that.

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)"

value "ch ≪''frank''≫"
value "sd ≪''frank''≫"

We are now able to define the ordering of strings.

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

instantiation str_calc :: (type) equiv
begin
definition equiv_str_calc :: "'a str_calc ⇒ 'a str_calc ⇒ bool" **where**
"equiv_str_calc p q ⟷ sd(q) = sd(p)"
instance
apply (intro_classes)
apply (unfold less_eq_str_calc_def less_str_calc_def equiv_str_calc_def)
apply (auto)
done
end

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
```

8.11   Note: Small interchange laws.

```
lemma empty_str_seq_unit:
"ε ; s ≅ s"
"s ; ε ≅ s"
apply (unfold equiv_str_calc_def)
apply (simp_all add: set_eq_iff)
done


lemma empty_str_par_unit:
"ε | s ≅ s"
"s | ε ≅ s"
apply (unfold equiv_str_calc_def)
apply (simp_all add: set_eq_iff)
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 less_str_calc_def)
apply (simp)
apply (auto)
done
```

8.12   Note: an example derivation

TODO: Construct a proof that replicates the step-wise calculation.

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