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

Tony Hoare Frank Zeyda Georg Struth

June 5, 2017

Abstract

Contents

1 The Option Monad: Supplement

```
theory Option_Monad
imports Eisbach "~~/src/HOL/Library/Monad_Syntax"
begin
```

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

1.1 Syntax and Definitions

The *return* function of the option monad (bind is already defined).

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

We use the notation ⊥ in place of `None`.

```
notation None ("⊥")
```

1.2 Proof Support

Proof support for reasoning about option types.

Attribute used to collection definitional laws for lifted operators.

```
named_theorems option_monad_ops
  "definitial laws for lifted operators into the option monad"
```

Tactic that performs automatic case splittings for the `option` type.

```
lemmas split_option =
  split_option_all split_option_ex

method option_tac = (
  (atomize (full))?,
  ((unfold option_monad_ops option_return_def)?) [1],
  (simp add: split_option)?)
end
```

2 Strict Operators

theory Strict_Operators
imports Main Real Option_Monad
begin

We encoded undefined values by virtue of the option monad.

Strict (lifted) operators always carry a subsection $_?$.

2.1 Equality

We define a strong notion of equality between undefined values.

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

2.2 Relational Operators

We also define lifted versions of the comparison operators in a similar way.

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

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

2.3 Multiplication and Division

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

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

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

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

```
declare lifted_times_def [option_monad_ops]
declare lifted_divide_def [option_monad_ops]
end
```

3 Examples of Applications

```
theory ICL
imports Main Real Strict_Operators
begin
```

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

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

3.1 Locale Definitions

In this section, we encapsulate the interchange law as an Isabelle locale. This gives us an elegant way to formulate conjectures that particular types, orderings, and operator pairs fulfill the interchange law. It is to some extent a design decision.

3.1.1 Locale: preorder

The underlying relation has to be a pre-order. Our definition of pre-order is, however, deliberately weaker than Isabelle/HOL's definition as per the `ordering` locale since we do not require the assumption `a < b ⟷ a ≤ b ∧ a ≠ b`. Moreover, for interpretation we only have to provide the ≤ operator in our treatment, but not <.

```
locale preorder =
— The type locale parameter is for convenience only.
  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 as mutual less-or-equals.

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

We prove that ≡ is indeed an equivalence relation.

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

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

lemma equiv_trans:
"x ≡ y ⟹ y ≡ z ⟹ x ≡ z"
apply (unfold equiv_def)
apply (clarsimp)
apply (rule conjI)
```

```
using local.trans apply (blast)
using local.trans apply (blast)
done
```

The following anti-symmetry law holds (by definition) as well.

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

3.1.2 Locale: iclaw

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

```
locale iclaw = preorder +
  fixes seq :: "'a ⇒ 'a ⇒ 'a" (infixr ";" 110)
  fixes par :: "'a ⇒ 'a ⇒ 'a" (infixr "|" 100)
  assumes interchange_law: "(p | r) ; (q | s) ≤ (p ; q) | (r ; s)"
```

3.2 Locale Interpretations

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

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

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

```
interpretation icl_plus_minus_nat:
  iclaw "TYPE(nat)" "op =" "op +" "op -"
apply (unfold_locales)
apply (auto) [2]
apply (linarith?)
oops

interpretation icl_plus_minus_int:
  iclaw "TYPE(int)" "op =" "op +" "op -"
apply (unfold_locales)
apply (auto) [2]
apply (linarith)
done

interpretation icl_plus_minus_rat:
  iclaw "TYPE(rat)" "op =" "op +" "op -"
apply (unfold_locales)
apply (auto) [2]
apply (linarith)
done

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

```
apply (unfold_locales)
apply (auto) [2]
apply (linarith)
done
```

3.2.2 Arithmetic: multiplication (x) and division (/).

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

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

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

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

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

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

3.2.3 Natural numbers: multiplication (x) and truncated division (div)

```
lemma trunc_div_lemma:
fixes p :: "nat"
fixes q :: "nat"
— assumes "q > 0" (not needed!)
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 (auto)
done
```

We note that Isabelle/HOL defines `x div 0 = 0`. Hence we can prove the law even in HOL's weak treatment of undefinedness. The law holds indeed in both the weak and strong treatment.

```
interpretation icl_mult_trunc_div_nat:
  iclaw "TYPE(nat)" "op ≤" "op *" "op div"
apply (unfold_locales)
apply (auto) [2]
apply (case_tac "r = 0"; simp_all)
apply (case_tac "s = 0"; simp_all)
apply (subgoal_tac "(p div r) * (q div s) * (r * s) ≤ p * q")
apply (metis div_le_mono div_mult_self_is_m nat_0_less_mult_iff)
```

```
apply (unfold semiring_normalization_rules(13))
apply (metis div_mult_self_is_m mult_le_mono trunc_div_lemma)
done

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

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

We can easily verify the definition of implication.

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

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

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

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

Numerical value of a boolean and the thus-induced ordering.

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

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

We can easily proof that the numerical order defined above is implication.

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

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

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