The Interchange Law in Application to Concurrent Programming

—

Mechanisation in Isabelle/HOL

Tony Hoare Frank Zeyda Georg Struth

June 27, 2017

Abstract

Contents

1 The Option Monad: Supplement **2**
1.1 Syntax and Definitions . . . 2
1.2 Proof Support . . . 2

2 The Overflow Monad **3**
2.1 Type Definition . . . 3
2.2 Ordering Relation . . . 3
2.3 Monadic Constructors . . . 3
2.4 Lifted Operators . . . 4
2.5 Proof Support . . . 4
2.6 Proof Experiments . . . 5

3 Strict Operators **6**
3.1 Equality . . . 6
3.2 Relational Operators . . . 6
3.3 Multiplication and Division . . . 6

4 Examples of Applications **8**
4.1 Locale Definitions . . . 8
4.1.1 Locale: `preorder` . . . 8
4.1.2 Locale: `iclaw` . . . 9
4.2 Locale Interpretations . . . 10
4.2.1 2. Arithmetic: addition (`+`) and subtraction (`-`) of numbers. . . . 10
4.2.2 3. Arithmetic: multiplication (`x`) and division (`/`). . . . 10
4.2.3 4. Natural numbers: multiplication (`x`) and truncated division (`-:-`) . . . 11
4.2.4 5. Propositional calculus: conjunction ($\wedge$) and implication ($\Rightarrow$). . . . 11
4.2.5 6. Boolean Algebra: conjunction ($\wedge$) and disjunction ($\vee$). . . . 12
4.2.6 7. Self-interchanging operators: `+`, `*`, $\vee$, $\wedge$. . . . 12
4.2.7 8. Computer arithmetic: Overflow ($\top$). . . . 13
4.2.8 9. Note: Partial operators. . . . 15

1 The Option Monad: Supplement

```
theory Option_Monad
imports "~~/src/HOL/Library/Monad_Syntax" Eisbach
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 The Overflow Monad

theory Overflow_Monad
imports Main "~~/src/HOL/Library/Monad_Syntax" Eisbach
begin

2.1 Type Definition

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

2.2 Ordering Relation

instantiation overflow :: (linorder) linorder
begin
fun less_eq_overflow :: "'a overflow ⇒ 'a overflow ⇒ bool" **where**
"Result x ≤ Result y ⟷ x ≤ y" |
"Result x ≤ Overflow ⟷ True" |
"Overflow ≤ Result x ⟷ False" |
"Overflow ≤ Overflow ⟷ True"
fun less_overflow :: "'a overflow ⇒ 'a overflow ⇒ bool" **where**
"Result x < Result y ⟷ x < y" |
"Result x < Overflow ⟷ True" |
"Overflow < Result x ⟷ False" |
"Overflow < Overflow ⟷ False"
instance
apply (intro_classes)
— Subgoal 1
apply (induct_tac x; induct_tac y; simp)
using le_less **apply** (auto) [1]
— Subgoal 2
apply (induct_tac x; simp)
— Subgoal 3
apply (unfold atomize_imp)
apply (induct_tac x; induct_tac y; induct_tac z; simp)
using order_trans **apply** (blast)
— Subgoal 4
apply (induct_tac x; induct_tac y; simp)
— Subgoal 5
apply (induct_tac x; induct_tac y; simp)
using le_cases **apply** (blast)
done
end

2.3 Monadic Constructors

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

adhoc_overloading
  bind overflow_bind

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

2.4 Lifted Operators

Attribute used to collection definitional laws for lifted operators.

```
named_theorems overflow_monad_ops
  "definitial laws for lifted operators into the overflow monad"

consts max_value :: "'a"
```

Tony, I think the definition below is not quite right! It is not just enough to check that `f x' y'` $\leq$ `max_value` since this does not imply that both `x'` $\leq$ `max_value` and `y'` $\leq$ `max_value`. Consider, for instance, `(0::'a) * y` $\leq$ `max_value` for any `y` (!).

```
definition check_overflow ::
  "('a::linorder ⇒ 'a ⇒ 'a) ⇒
   ('a overflow ⇒ 'a overflow ⇒ 'a overflow)" where
[overflow_monad_ops]: "check_overflow f x y =
  do {x' ← x; y' ← y; if (f x' y') ≤ max_value then return (f x' y') else ⊤}"

definition overflow_times ::
  "'a::{times,linorder} overflow ⇒ 'a overflow ⇒ 'a overflow"
  (infixl "[*]" 70) where
[overflow_monad_ops]: "overflow_times = check_overflow (op *)"

definition overflow_divide ::
  "'a::{divide,linorder} overflow ⇒ 'a overflow ⇒ 'a overflow"
  (infixl "[div]" 70) where
[overflow_monad_ops]: "overflow_divide = check_overflow (op div)"
```

2.5 Proof Support

Proof support for reasoning about `overflow` types.

```
lemma split_overflow_all:
"(∀x::'a::linorder overflow. P x) = (P Overflow ∧ (∀x::'a. P (Result x)))"
apply (safe; simp?)
apply (case_tac x)
apply (simp_all)
done

lemma split_overflow_ex:
"(∃x::'a::linorder overflow. P x) = (P Overflow ∨ (∃x::'a. P (Result x)))"
apply (safe; simp?)
apply (case_tac x)
apply (simp_all) [2]
apply (auto)
done
```

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

```
lemmas split_overflow =
  split_overflow_all split_overflow_ex

method overflow_tac = (
  (atomize (full))?,
  ((unfold overflow_monad_ops overflow_return_def)?) [1],
  (simp add: split_overflow)?)
```

2.6 Proof Experiments

lemma "∀(x::nat overflow) y. x [*] y = y [*] x"
apply (overflow_tac)
apply (simp add: semiring_normalization_rules(7))
done
end

3 Strict Operators

theory Strict_Operators
imports Main Real Option_Monad Option
  "~~/src/HOL/Library/Option_ord"
begin

We encoded undefined values by virtue of the option monad.

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

3.1 Equality

We define a strong notion of equality between undefined values.

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

3.2 Relational Operators

We also define lifted versions of 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 ⟷ True" |
"None ≤$_?$ None ⟷ True"

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

The above definitions coincide with the default ordering on option.

lemma lifted_leq_equiv_option_ord:
"op ≤$_?$ = op ≤"
apply (rule ext)+
apply (option_tac)
done

lemma lifted_less_equiv_option_ord:
"op <$_?$ = op <"
apply (rule ext)+
apply (option_tac)
done

3.3 Multiplication and Division

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

4 Examples of Applications

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

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

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

4.1 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 triples fulfill the interchange law. It also aids in structuring proofs. We define two locales here: one to introduce the notion of order (which has to be a preorder); and another locale that extends the former and introduces both operators for the interchange law to hold.

4.1.1 Locale: `preorder`

The underlying relation has to be a preorder. Our definition of preorder is, however, deliberately weaker than Isabelle/HOL's definition as per the `ordering` locale. This is because we do not require the assumption `a < b` $\longleftrightarrow$ `a` $\leq$ `b` $\wedge$ `a` $\neq$ `b`. Moreover, for interpretation we only have to provide the $\leq$ operator in our treatment and not `<` also.

```
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 as mutual less-or-equals.

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

We prove that $\equiv$ 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
```

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
```

Next, we prove several instantiations of `orderings` used later on. Due to the structuring of locales, we will be able to use those lemmas in instantiations proofs of the `iclaw` locales which we define in the sequel.

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

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

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

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

Make the above instantiation lemmas automatic simplifications.

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

4.1.2 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 sequence/parallel operator pair.

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

4.2 Locale Interpretations

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

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

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

```
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
```

4.2.2 3. 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
```

4.2.3 4. Natural numbers: multiplication (x) and truncated division (-:-)

We note that the operator div is used in Isabelle for truncated division.

```
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, as well as the stronger one.

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

interpretation icl_mult_trunc_div_nat_strong:
  iclaw "TYPE(nat option)" "op ≤?" "op *?" "op /?"
apply (unfold_locales)
apply (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
```

4.2.4 5. Propositional calculus: conjunction (∧) and implication (⇒).

We can easily verify the definition of implication.

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

done

We note that $\vdash$ is encoded by object-logic implication ($\longrightarrow$).

```
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
```

4.2.5 6. Boolean Algebra: conjunction ($\wedge$) and disjunction ($\vee$).

Numerical value of a boolean and the thereby 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 show 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)" "numOrdBool" "op ∨" "op ∧"
apply (unfold_locales)
apply (unfold numOrdBool_is_imp)
apply (auto)
done
```

4.2.6 7. Self-interchanging operators: +, *, $\vee$, $\wedge$.

We introduce individual locales to capture associativity, commutativity and idempotence of a binary operator.

```
no_notation comp (infixl "∘" 55)

locale assoc =
  fixes operator :: "'a ⇒ 'a ⇒ 'a" (infix "∘" 100)
  assumes assoc: "x ∘ (y ∘ z) = (x ∘ y) ∘ z"

locale comm =
  fixes operator :: "'a ⇒ 'a ⇒ 'a" (infix "∘" 100)
  assumes comm: "x ∘ y = y ∘ x"

locale unit =
  fixes operator :: "'a ⇒ 'a ⇒ 'a" (infix "∘" 100)
  fixes unit :: "'a" ("1")
```

```
  assumes left_unit [simp]: "1 ∘ x = x"
  assumes right_unit [simp]: "x ∘ 1 = x"

declare unit.left_unit [simp]
declare unit.right_unit [simp]

lemma assoc_comm_imp_iclaw:
"(assoc bop ∧ comm bop) ⟹ (iclaw (op =) bop bop)"
apply (unfold iclaw_def iclaw_axioms_def)
apply (clarsimp)
apply (unfold assoc_def comm_def iclaw_def)
apply (clarsimp)
done

lemma iclaw_unit_imp_assoc_comm:
"(iclaw (op =) bop bop) ∧ (unit bop one) ⟹ (assoc bop ∧ comm bop)"
apply (unfold iclaw_def iclaw_axioms_def)
apply (clarsimp)
apply (rule conjI)
— Subgoal 1
apply (unfold assoc_def)
apply (clarify)
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 (clarsimp)
— Subgoal 1
apply (unfold comm_def)
apply (clarify)
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 (clarsimp)
done
```

4.2.7 8. Computer arithmetic: Overflow ($\top$).

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

declare [[syntax_ambiguity_warning = false]]

definition comparith_times ::
  "'a::{times,linorder} comparith ⇒ 'a comparith ⇒ 'a comparith"
    (infixl "*c" 70) where
[option_monad_ops]: "x *c y = do {x' ← x; y' ← y; return (x' [*] y')}"

definition comparith_divide ::
  "'a::{zero,divide,linorder} comparith ⇒ 'a comparith ⇒ 'a comparith"
    (infixl "'/c" 70) where
[option_monad_ops]: "x /c y =
  do {x' ← x; y' ← y; if y' ≠ Result 0 then return (x' [div] y') else ⊥}"

lemma check_overflow_simps [simp]:
"check_overflow f x ⊤ = ⊤"
```

```
"check_overflow f ⊤ y = ⊤"
apply (unfold check_overflow_def)
apply (case_tac x; simp)
apply (case_tac y; simp)
done

lemma check_overflow_Result:
"check_overflow f (Result x) (Result y) =
  (if (f x y) ≤ max_value then Result (f x y) else ⊤)"
apply (unfold check_overflow_def)
apply (case_tac "(f x y) ≤ max_value")
apply (simp_all)
done

lemma check_overflow_neq_Result_0:
"(x::nat overflow) ≠ Result 0 ⟹
 (y::nat overflow) ≠ Result 0 ⟹
 (check_overflow op * x y ≠ Result 0)"
apply (unfold check_overflow_def)
apply (case_tac x; case_tac y)
apply (simp_all)
done

lemma section_4_cancal_law1:
"p ≤ (p [*] q) [div] q"
apply (unfold overflow_times_def overflow_divide_def)
apply (induct_tac p; induct_tac q)
apply (simp_all)
apply (rename_tac p q)
apply (simp add: check_overflow_Result)
apply (clarsimp)
oops

interpretation icl_mult_trunc_div_nat_overflow:
  iclaw "TYPE(nat comparith)" "op ≤" "op *c" "op /c"
apply (unfold_locales)
apply (option_tac)
apply (unfold overflow_times_def overflow_divide_def)
apply (simp add: check_overflow_neq_Result_0)
apply (clarify)
apply (thin_tac "r ≠ Result 0")
apply (thin_tac "s ≠ Result 0")

apply (induct_tac p; induct_tac r; induct_tac q; induct_tac s)
apply (simp_all)
apply (rename_tac p r q s)
apply (unfold check_overflow_Result)
apply (case_tac "p div r ≤ max_value"; case_tac "q div s ≤ max_value";
       case_tac "p * q ≤ max_value"; case_tac "r * s ≤ max_value")
apply (simp_all)
apply (unfold check_overflow_Result)
apply (simp_all)
— Subgoal 1
using icl_mult_trunc_div_nat.interchange_law order_trans apply (blast)
— Subgoal 2
```

apply (erule_tac Q = "¬ q div s ≤ max_value" **in** contrapos_pp)
apply (clarsimp)
— We can see that this is not provable if p = 0, r = 1 and s = 1.

apply (subgoal_tac "p ≠ 0")
apply (metis div_le_dividend less_le_trans nonzero_mult_div_cancel_left not_le)
defer
— Subgoal 3
apply (erule_tac Q = "¬ p div r ≤ max_value" **in** contrapos_pp)
apply (clarsimp)
apply (subgoal_tac "q ≠ 0")
apply (metis div_le_dividend dual_order.trans nonzero_mult_div_cancel_right)
defer
— Subgoal 4
apply (metis div_le_dividend mult_zero_right nonzero_mult_div_cancel_right order_trans)
oops

4.2.8 9. Note: Partial operators.

As already explained and used above, the lifting of total into partial operators is achieved with the `option` types.

end