# Monad Normalisation

Joshua Schneider and Manuel Eberl and Andreas Lochbihler

September 23, 2023

**Abstract**

The usual monad laws can directly be used as rewrite rules for Isabelle's simplifier to normalise monadic HOL terms and decide equivalences. In a commutative monad, however, the commutativity law is a higher-order permutative rewrite rule that makes the simplifier loop. This AFP entry implements a simproc that normalises monadic expressions in commutative monads using ordered rewriting. The simproc can also permute computations across control operators like *if* and *case*.

## Contents



## 1 Normalisation of monadic expressions

**theory** *Monad-Normalisation*
**imports** *HOL−Probability.Probability*
**keywords** *print-monad-rules* :: *diag*
**begin**

The standard monad laws

$$return\ a \gg\!= f = f\ a \tag{1}$$

$$x \gg\!= return = x \tag{2}$$

$$(x \gg\!= f) \gg\!= g = x \gg\!= (\lambda a.\ f\ a \gg\!= g) \tag{3}$$

yield a confluent and terminating rewrite system. Thus, when these equations are added to the simpset, the simplifier can normalise monadic expressions and decide the equivalence of monadic programs (in the free monad).

However, many monads satisfy additional laws. In some monads, it is possible to discard unused effects

$$x \gg\!= (\lambda\text{-}.\ y) = y \tag{4}$$

or duplicate effects

$$x \gg\!= (\lambda a.\ x \gg\!= f\ a) = x \gg\!= (\lambda a.\ f\ a\ a) \tag{5}$$

or commute independent computations

$$x \gg\!= (\lambda a.\ y \gg\!= f\ a) = y \gg\!= (\lambda b.\ x \gg\!= (\lambda a.\ f\ a\ b)). \tag{6}$$

For example, - *option* satisfies (5) and (6), - *set* validates (6), and - *pmf* satisfies (4) and (6). Equations (4) and (5) can be directly used as rewrite rules.[1] However, the simplifier does not handle (6) well because (6) is a higher-order permutative rewrite rule and ordered rewriting in the simplifier can only handle first-order permutative rewrite rules. Consequently, when (6) is added to the simpset, the simplifier will loop.

This AFP entry implements a simproc for the simplifier to simplify HOL expressions over commutative monads based on ordered rewriting. This yields a decision procedure for equality of monadic expressions in any (free) monad satisfying any of the laws (4–6). If further commutative operators show up in the HOL term, then the ordered rewrite system need not be confluent and the simproc only performs a best effort. We do not know whether this general case can be solved by ordered rewriting as a complete solution would have to solve the graph isomorphism problem by term rewriting [1].

⟨*ML*⟩

## 1.1 Usage

The monad laws (1), (2), (3), (6) must be registered with the attribute *monad-rule*. The simproc does not need (4) and (5), so these properties need not be registered, but can simply be added to the simpset if needed. The simproc is activated by including the bundle *monad-normalisation*.

Additionally, distributivity laws for control operators like *If* and *case-nat* specialised to $\gg\!=$ can be declared with the attribute *monad-distrib*. Then, the simproc will also commute computations over these control operators.

The set of registered monad laws can be inspected with the command **print-monad-rules**.

[1] If they both hold, then (6) holds, too [2].

## 1.2 Registration of the monads from the Isabelle/HOL library

**lemmas** [*monad-rule*] = *Set.bind-bind*

**lemma** *set-bind-commute* [*monad-rule*]:
**fixes** $A :: 'a\ set$ **and** $B :: 'b\ set$
**shows** $A \gg\!= (\lambda x.\ B \gg\!= C\ x) = B \gg\!= (\lambda y.\ A \gg\!= (\lambda x.\ C\ x\ y))$
⟨*proof*⟩

**lemma** *set-return-bind* [*monad-rule*]:
**fixes** $A :: 'a \Rightarrow 'b\ set$
**shows** $\{x\} \gg\!= A = A\ x$
⟨*proof*⟩

**lemma** *set-bind-return* [*monad-rule*]:
**fixes** $A :: 'a\ set$
**shows** $A \gg\!= (\lambda x.\ \{x\}) = A$
⟨*proof*⟩

**lemmas** [*monad-rule*] = *Predicate.bind-bind Predicate.bind-single Predicate.single-bind*

**lemma** *predicate-bind-commute* [*monad-rule*]:
**fixes** $A :: 'a\ Predicate.pred$ **and** $B :: 'b\ Predicate.pred$
**shows** $A \gg\!= (\lambda x.\ B \gg\!= C\ x) = B \gg\!= (\lambda y.\ A \gg\!= (\lambda x.\ C\ x\ y))$
⟨*proof*⟩

**lemmas** [*monad-rule*] = *Option.bind-assoc Option.bind-lunit Option.bind-runit*

**lemma** *option-bind-commute* [*monad-rule*]:
**fixes** $A :: 'a\ option$ **and** $B :: 'b\ option$
**shows** $A \gg\!= (\lambda x.\ B \gg\!= C\ x) = B \gg\!= (\lambda y.\ A \gg\!= (\lambda x.\ C\ x\ y))$
⟨*proof*⟩

**lemmas** [*monad-rule*] =
*bind-assoc-pmf*
*bind-commute-pmf*
*bind-return-pmf*
*bind-return-pmf'*

**lemmas** [*monad-rule*] =
*bind-spmf-assoc*
*bind-commute-spmf*
*bind-return-spmf*
*return-bind-spmf*

## 1.3 Distributive operators

**lemma** *bind-if* [*monad-distrib*]:

*f A* (λ*x*. *if P then B x else C x*) = (*if P then f A B else f A C*)
⟨*proof*⟩

**lemma** *bind-case-prod* [*monad-distrib*]:
*f A* (λ*x*. *case y of* (*a*,*b*) ⇒ *B a b x*) = (*case y of* (*a*,*b*) ⇒ *f A* (*B a b*))
⟨*proof*⟩

**lemma** *bind-case-sum* [*monad-distrib*]:
*f A* (λ*x*. *case y of Inl a* ⇒ *B a x* | *Inr a* ⇒ *C a x*) =
(*case y of Inl a* ⇒ *f A* (*B a*) | *Inr a* ⇒ *f A* (*C a*))
⟨*proof*⟩

**lemma** *bind-case-option* [*monad-distrib*]:
*f A* (λ*x*. *case y of Some a* ⇒ *B a x* | *None* ⇒ *C x*) =
(*case y of Some a* ⇒ *f A* (*B a*) | *None* ⇒ *f A C*)
⟨*proof*⟩

**lemma** *bind-case-list* [*monad-distrib*]:
*f A* (λ*x*. *case y of* [] ⇒ *B x* | *y* # *ys* ⇒ *C y ys x*) = (*case y of* [] ⇒ *f A B* | *y* # *ys* ⇒ *f A* (*C y ys*))
⟨*proof*⟩

**lemma** *bind-case-nat* [*monad-distrib*]:
*f A* (λ*x*. *case y of 0* ⇒ *B x* | *Suc n* ⇒ *C n x*) = (*case y of 0* ⇒ *f A B* | *Suc n* ⇒ *f A* (*C n*))
⟨*proof*⟩

### 1.4 Setup of the normalisation simproc

⟨*ML*⟩
**declare** [[*simproc del*: *monad-normalisation*]]

The following bundle enables normalisation of monadic expressions by the simplifier. We use *monad-rule-internal* instead of *monad-rule*[*simp*] such that the theorems in *monad-rule* get dynamically added to the simpset instead of only those that are in there when the bundle is declared.

**bundle** *monad-normalisation* = [[*simproc add*: *monad-normalisation*, *monad-rule-internal*]]

**end**

**theory** *Monad-Normalisation-Test*
**imports** *Monad-Normalisation*
**begin**

## 2 Tests and examples

**context includes** *monad-normalisation*

**begin**

**lemma**
**assumes** $f = id$
**shows**
$do\ \{x \leftarrow B;\ z \leftarrow C\ x;\ d \leftarrow E\ z\ x;\ a \leftarrow D\ z\ x;\ y \leftarrow A;\ return\text{-}pmf\ (x,y)\} =$
$do\ \{y \leftarrow A;\ x \leftarrow B;\ z \leftarrow C\ x;\ a \leftarrow D\ z\ x;\ d \leftarrow E\ z\ x;\ return\text{-}pmf\ (f\ (x,y))\}$
⟨*proof*⟩

**lemma** $(do\ \{a \leftarrow E;\ b \leftarrow E;\ w \leftarrow B\ b\ a;\ z \leftarrow B\ a\ b;\ return\text{-}pmf\ (w,z)\}) =$
$(do\ \{a \leftarrow E;\ b \leftarrow E;\ z \leftarrow B\ a\ b;\ w \leftarrow B\ b\ a;\ return\text{-}pmf\ (w,z)\})$
⟨*proof*⟩

**lemma** $(do\ \{a \leftarrow E;\ b \leftarrow E;\ w \leftarrow B\ b\ a;\ z \leftarrow B\ a\ b;\ return\text{-}pmf\ (w,z)\}) =$
$(do\ \{a \leftarrow E;\ b \leftarrow E;\ z \leftarrow B\ a\ b;\ w \leftarrow B\ b\ a;\ return\text{-}pmf\ (w,z)\})$
⟨*proof*⟩

**lemma** $do\ \{y \leftarrow A;\ x \leftarrow A;\ z \leftarrow B\ x\ y\ y;\ w \leftarrow B\ x\ x\ y;\ Some\ (x,y)\} =$
$do\ \{x \leftarrow A;\ y \leftarrow A;\ z \leftarrow B\ x\ x\ y;\ w \leftarrow B\ x\ y\ y;\ Some\ (x,y)\}$
⟨*proof*⟩

**lemma** $do\ \{y \leftarrow A;\ x \leftarrow A;\ z \leftarrow B\ x\ y\ y;\ w \leftarrow B\ x\ x\ y;\ \{x,y\}\} =$
$do\ \{x \leftarrow A;\ y \leftarrow A;\ z \leftarrow B\ x\ x\ y;\ w \leftarrow B\ x\ y\ y;\ \{x,y\}\}$
⟨*proof*⟩

**lemma** $do\ \{y \leftarrow A;\ x \leftarrow A;\ z \leftarrow B\ x\ y\ y;\ w \leftarrow B\ x\ x\ y;\ return\text{-}pmf\ (x,y)\} =$
$do\ \{x \leftarrow A;\ y \leftarrow A;\ z \leftarrow B\ x\ x\ y;\ w \leftarrow B\ x\ y\ y;\ return\text{-}pmf\ (x,y)\}$
⟨*proof*⟩

**lemma** $do\ \{x \leftarrow A\ 0;\ y \leftarrow A\ x;\ w \leftarrow B\ y\ y;\ z \leftarrow B\ x\ y;\ a \leftarrow C;\ Predicate.single\ (a,a)\} =$
$do\ \{x \leftarrow A\ 0;\ y \leftarrow A\ x;\ z \leftarrow B\ x\ y;\ w \leftarrow B\ y\ y;\ a \leftarrow C;\ Predicate.single\ (a,a)\}$
⟨*proof*⟩

**lemma** $do\ \{x \leftarrow A\ 0;\ y \leftarrow A\ x;\ z \leftarrow B\ x\ y;\ w \leftarrow B\ y\ y;\ a \leftarrow C;\ return\text{-}pmf\ (a,a)\} =$
$do\ \{x \leftarrow A\ 0;\ y \leftarrow A\ x;\ z \leftarrow B\ y\ y;\ w \leftarrow B\ x\ y;\ a \leftarrow C;\ return\text{-}pmf\ (a,a)\}$
⟨*proof*⟩

**lemma** $do\ \{x \leftarrow B;\ z \leftarrow C\ x;\ d \leftarrow E\ z\ x;\ a \leftarrow D\ z\ x;\ y \leftarrow A;\ return\text{-}pmf\ (x,y)\} =$
$do\ \{y \leftarrow A;\ x \leftarrow B;\ z \leftarrow C\ x;\ a \leftarrow D\ z\ x;\ d \leftarrow E\ z\ x;\ return\text{-}pmf\ (x,y)\}$
⟨*proof*⟩

**no-adhoc-overloading** *Monad-Syntax.bind bind-pmf*

**context**

**fixes** $\mathcal{A}1 :: 'a \Rightarrow (('a \times 'a) \times 'b)\ spmf$
**and** $\mathcal{A}2 :: 'a \times 'a \Rightarrow 'b \Rightarrow bool\ spmf$
**and** $sample\text{-}uniform :: nat \Rightarrow nat\ spmf$
**and** $order :: 'a \Rightarrow nat$
**begin**

**lemma**
$do\ \{$
$x \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$y \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$z \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$b \leftarrow coin\text{-}spmf$;
$((msg1,\ msg2),\ \sigma) \leftarrow \mathcal{A}1\ (f\ x)$;
$\text{-} :: unit \leftarrow assert\text{-}spmf\ (valid\text{-}plain\ msg1 \wedge valid\text{-}plain\ msg2)$;
$guess \leftarrow \mathcal{A}2\ (f\ y,\ xor\ (f\ z)\ (if\ b\ then\ msg1\ else\ msg2))\ \sigma$;
$return\text{-}spmf\ (guess \longleftrightarrow b)$
$\} = do\ \{$
$x \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$y \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$((msg1,\ msg2),\ \sigma) \leftarrow \mathcal{A}1\ (f\ x)$;
$\text{-} :: unit \leftarrow assert\text{-}spmf\ (valid\text{-}plain\ msg1 \wedge valid\text{-}plain\ msg2)$;
$b \leftarrow coin\text{-}spmf$;
$x \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$guess \leftarrow \mathcal{A}2\ (f\ y,\ xor\ (f\ x)\ (if\ b\ then\ msg1\ else\ msg2))\ \sigma$;
$return\text{-}spmf\ (guess \longleftrightarrow b)$
$\}$ **for** $xor$
⟨*proof*⟩

**lemma**
$do\ \{$
$x \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$xa \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$x \leftarrow \mathcal{A}1\ (f\ x)$;
$case\ x\ of$
$(x,\ xb) \Rightarrow$
$(case\ x\ of$
$(msg1,\ msg2) \Rightarrow$
$\lambda\sigma.\ do\ \{$
$a \leftarrow assert\text{-}spmf\ (valid\text{-}plain\ msg1 \wedge valid\text{-}plain\ msg2)$;
$x \leftarrow coin\text{-}spmf$;
$xaa \leftarrow map\text{-}spmf\ f\ (sample\text{-}uniform\ (order\ \mathcal{G}))$;
$guess \leftarrow \mathcal{A}2\ (f\ xa,\ xaa)\ \sigma$;
$return\text{-}spmf\ (guess \longleftrightarrow x)$
$\})$
$xb$
$\} = do\ \{$
$x \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$xa \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
$x \leftarrow \mathcal{A}1\ (f\ x)$;

```
    case x of
    (x, xb) ⇒
      (case x of
       (msg1, msg2) ⇒
         λσ. do {
              a ← assert-spmf (valid-plain msg1 ∧ valid-plain msg2);
              z ← map-spmf f (sample-uniform (order 𝒢));
              guess ← 𝒜2 (f xa, z) σ;
              map-spmf ((⟷) guess) coin-spmf
            })
        xb
  }
⟨proof⟩

lemma elgamal-step3:
  do {
    x ← sample-uniform (order 𝒢);
    y ← sample-uniform (order 𝒢);
    b ← coin-spmf;
    p ← 𝒜1 (f x);
    - ← assert-spmf (valid-plain (fst (fst p)) ∧ valid-plain (snd (fst p)));
    guess ←
      𝒜2 (f y, xor (f (x * y)) (if b then fst (fst p) else snd (fst p)))
       (snd p);
    return-spmf (guess ⟷ b)
  } = do {
    y ← sample-uniform (order 𝒢);
    b ← coin-spmf;
    p ← 𝒜1 (f y);
    - ← assert-spmf (valid-plain (fst (fst p)) ∧ valid-plain (snd (fst p)));
    ya ← sample-uniform (order 𝒢);
    b' ← 𝒜2 (f ya,
              xor (f (y * ya)) (if b then fst (fst p) else snd (fst p)))
          (snd p);
    return-spmf (b' ⟷ b)
  } for xor
⟨proof⟩

end
```

Distributivity

```
lemma
  do {
    x ← A :: nat spmf;
    a ← B;
    b ← B;
    if a = b then do {
      return-spmf x
    } else do {
```

```
    y ← C;
    return-spmf (x + y)
  }
} = do {
  a ← B;
  b ← B;
  if b = a then A else do {
    y ← C;
    x ← A;
    return-spmf (y + x)
  }
}
⟨proof⟩

lemma
  do {
    x ← A :: nat spmf;
    p ← do {
      a ← B;
      b ← B;
      return-spmf (a, b)
    };
    q ← coin-spmf;
    if q then do {
      return-spmf (x + fst p)
    } else do {
      y ← C;
      return-spmf (y + snd p)
    }
  } = do {
    q ← coin-spmf;
    if q then do {
      x ← A;
      a ← B;
      - ← B;
      return-spmf (x + a)
    } else do {
      y ← C;
      a ← B;
      - ← B;
      - ← A;
      return-spmf (y + a)
    }
  }
⟨proof⟩

lemma
  fixes f :: nat ⇒ nat ⇒ nat + nat
  shows
```

```
  do {
     x ← (A::nat set);
     a ← B;
     b ← B;
     case f a b of
       Inl c ⇒ {x}
     | Inr c ⇒ do {
         y ← C x;
         {(x + y + c)}
       }
  } = do {
     a ← B;
     b ← B;
     case f b a of
       Inl c ⇒ A
     | Inr c ⇒ do {
         x ← A;
         y ← C x;
         {(y + c + x)}
     }
  }
⟨proof⟩
```

# 3 Limits

The following example shows that the combination of monad normalisation and regular ordered rewriting is not necessarily confluent.

```
lemma do {a ← A; b ← A; Some (a ∧ b, b)} =
      do {a ← A; b ← A; Some (a ∧ b, a)}
⟨proof⟩
```

The next example shows that even monad normalisation alone is not confluent because the term ordering prevents the reordering of $f\ A$ with $f\ B$. But if we change $A$ to $E$, then the reordering works as expected.

```
lemma
  do {a ← f A; b ← f B; c ← D b; d ← f C; F a c d} =
   do {b ← f B; c ← D b; a ← f A; d ← f C; F a c d}
  for f :: 'b ⇒ 'a option and D :: 'a ⇒ 'a option
  ⟨proof⟩

lemma
  do {a ← f E; b ← f B; c ← D b; d ← f C; F a c d} =
   do {b ← f B; c ← D b; a ← f E; d ← f C; F a c d}
  for f :: 'b ⇒ 'a option and D :: 'a ⇒ 'a option
  ⟨proof⟩

end
```

**end**

## References


[1] D. A. Basin. A term equality problem equivalent to graph isomorphism. *Information Processing Letters*, 51:61–66, 1994.

[2] A. Lochbihler and J. Schneider. Equational reasoning with applicative functors. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *LNCS*, pages 252–273. Springer, 2016.