Monadic equational reasoning for `while` loop in Rocq

Ryuji Kawakami [1], Jacques Garrigue [2], Takahumi Saikawa [2], Reynald Affeldt [3]

[1]SOKENDAI, Japan
[2]Nagoya University, Japan
[3]National Institute of Advanced Industrial Science and Technology (AIST), Japan

While statement in Rocq

- Rocq does not permit the definition of non-terminating functions.

```
(* OCaml code *)
let rec collatz x =
  if x <= 1 then x else
  if x mod 2 = 0
  then collatz (x / 2)
  else collatz (3 * x + 1)
```

`Fixpoint` Command	×
`Equation` Command	×
`CoFixpoint` Command	△

Our Idea

Execute while by monadic rewriting! (and use coinduction under the hood)

Monad: A method for expressing computational effects

- A monad consists of a functor M and two operations: Ret and Bind

$$\begin{aligned} \mathsf{M} &: \mathsf{Type} \to \mathsf{Type} \\ \mathsf{Ret} &: \mathsf{A} \to \mathsf{M\ A} \\ \gg= &: \mathsf{M\ A} \to (\mathsf{A} \to \mathsf{M\ B}) \to \mathsf{M\ B} \end{aligned}$$

- In functional programming, monads are used to express computational effects.
- The following three axioms are satisfied by Ret and $\gg=$:

Monad laws

$$\begin{aligned} &\mathsf{bindretf} : \mathsf{Ret\ a} \gg= \mathsf{f} = \mathsf{f\ (a)} \\ &\mathsf{bindmret} : \mathsf{m} \gg= \mathsf{Ret} = \mathsf{m} \\ &\mathsf{bindA} : (\mathsf{m} \gg= \mathsf{f}) \gg= \mathsf{g} = \mathsf{m} \gg= (\lambda x.\ \mathsf{f\ (x)} \gg= \mathsf{g}) \end{aligned}$$

Monae: monadic equational reasoning in Rocq [ANS19]

- Monadic equational reasoning [GH11]
- Monae already supports probability monad, non-determinism monad, etc.

Interfaces: For equational reasoning

```
HB.mixin Record isMonadState (S : Type) (M : Type -> Type) of Monad M :=
{ get : M S ;
  put : S -> M unit ;
  putput : forall s s', put s >> put s' = put s' ;
  putget : forall s, put s >> get = put s >> Ret s ;
  ...
```

Models: For soundness only (implementations are hiden to user).

```
Definition M := fun A : Type => S -> A * S.
...
Let get : M S := fun s => (s, s).
Let put : S -> M unit := fun s => fun s' => (tt, s').
Let putput : forall s s', put s >> put s' = put s'. Proof. by []. Qed.
Let putget : forall s, put s >> get = put s >> Ret s. ...
```

Overview

Our contribution

We extend the Monae library to express while loops.

- We can write proofs using the `rewrite` tactic.
- We can combine some other effects using monad transformers.

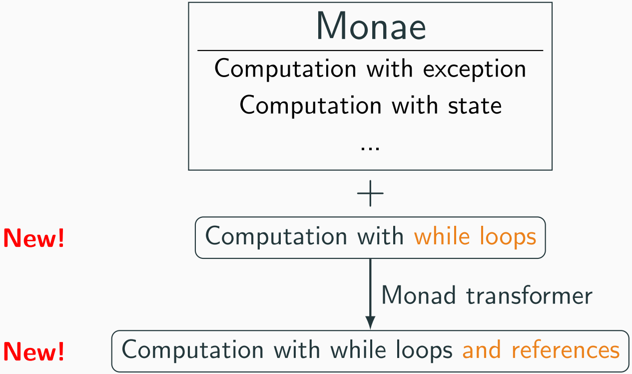


Illustration: factorial using a while loop and references

```
while : (X -> M (A + X)) -> X -> M A
```

```
let fact n =
let r = ref 1 in
let l = ref 1 in




while !l <= n do


  r := !r * !l;
  l := !l + 1;

done;
!r
```

⇛

```
Definition factdts n :=
do r <- cnew ml_int 1;
do l <- cnew ml_int 1;
do _ <-
while (fun (_ : unit) =>
        do i <- cget l;
        if i <= n
        then do v <- cget r;
             do _ <- cput r (i * v);
             do _ <- cput l (i.+1);
             Ret (inr tt)
        else Ret (inl tt)) tt;
do v <- cget r; Ret v.
```

Interface: the complete Elgot monad i

- The Complete Elgot monad [AMV10] treats recursive structures algebraically.
- An iteration operator

```
while : (X -> M (A + X)) -> X -> M A
```

- Equations: `fixpointE`, `naturalityE`, `codiagonalE`, `uniformE`

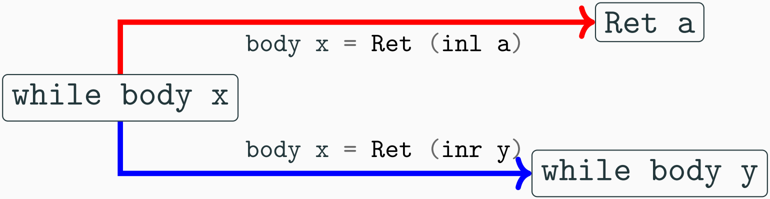
```
fixpointE : forall (f : A -> M (B + A)) (a : A),
  while f a ≈ f a >>= sum_rect Ret (while f)
naturalityE : forall (f : A -> M (B + A)) (g : B -> M C) (a : A),
  while f a >>= g ≈
  while (fun y => f y >>= sum_rect (M # inl \o g)
                                   (M # inr \o Ret)) a
codiagonalE : forall (f : A -> M ((B + A) + A)) (a : A),
  while ((M # ((sum_rect (fun => (B + A)) idfun inr))) \o f) a
  ≈ while (while f) a
```

Interface: the complete Elgot monad ii

```
uniformE :
  forall (f : A -> M (B + A)) (g : C -> M (B + C)) (h : C -> A),
  (forall c, f (h c) = g c >>= sum_rect ((M # inl) \o Ret)
                                         ((M # inr) \o Ret \o h))
  -> forall c, while f (h c) ≈ while g c
```

- fixpointE: loop unrolling.

```
fixpointE : forall (body : X -> M (A + X)) (x : X),
  while body x = (body x) >>= (sum_rect Ret (while body))
```



Reminder: McCarthy's 91 function

- For any input n $\leq$ 101, McCarthy's 91 function returns 91.

```
(* OCaml definition *)
let rec mc91 n = if 100 < n then n - 10 else mc91 (mc91 (n + 11))
```

Calculation of `mc91`

```
mc91(98)
= mc91(mc91(109))
= mc91(99)
= mc91(mc91(110))
= mc91(100)
= mc91(mc91(111))
= mc91(101)
= 91
```

- Rocq cannot define this function structurally due to the nested recursion.

McCarthy's 91 function in Rocq

- `n` : depth of recursion, `m` : value

In C

```
int mc91 (int n, int m) {
    while (n != 0) {
        if (m > 100) {
            n -= 1;
            m -= 10;
        } else {
            n += 1;
            m += 11;
        }
    }
    return m;}
```

⟹

In Rocq

```
Let mc91_body nm :=
  if nm.1 == 0 then Ret (inl nm.2)
  else if nm.2 > 100
       then Ret (inr (nm.1 - 1, nm.2 - 10))
       else Ret (inr (nm.1 + 1, nm.2 + 11)).

Let mc91 n m := while mc91_body (n + 1, m).
```

Proof using equational reasoning

- We proved $89 < m < 101 \Rightarrow mc91\ (m) = mc91\ (m + 1)$.

`bindretf`	`Ret a >>= f = f a`
`fixpointE`	`while f a = (f a) >>= (sum_rect Ret (while f))`

```
mc91 n m
```

Proof using equational reasoning

- We proved $89 < m < 101 \Rightarrow mc91\ (m) = mc91\ (m+1)$.

`bindretf`	`Ret a >>= f = f a`
`fixpointE`	`while f a = (f a) >>= (sum_rect Ret (while f))`

```
mc91 n m
= 《 definition of mc91 》
while (fun nm => if nm.1 == 0
                 then Ret (inl nm.2)
                 else if 100 < nm.1
                      then Ret (inr (nm.1.-1, nm.2 - 10))
                      else Ret (inr (nm.1.+1, nm.2 + 11))) (n.+1, m)
```

Proof using equational reasoning

- We proved $89 < m < 101 \Rightarrow mc91\ (m) = mc91\ (m + 1)$.

`bindretf`	`Ret a >>= f = f a`
`fixpointE`	`while f a = (f a) >>= (sum_rect Ret (while f))`

```
mc91 n m
=  《 definition of mc91 》
while (fun nm => if nm.1 == 0
                 then Ret (inl nm.2)
                 else if 100 < nm.1
                      then Ret (inr (nm.1.-1, nm.2 - 10))
                      else Ret (inr (nm.1.+1, nm.2 + 11))) (n.+1, m)
《 fixpointE 》
= (if 100 < m
   then Ret (inr (n, m - 10))
   else Ret (inr (n.+2, m + 11))) >>= sum_rect Ret (while mc91_body)
```

Proof using equational reasoning

- We proved $89 < m < 101 \Rightarrow mc91\ (m) = mc91\ (m + 1)$.

bindretf	Ret a >>= f = f a
fixpointE	while f a = (f a) >>= (sum_rect Ret (while f))

```
mc91 n m
= 《 definition of mc91 》
while (fun nm => if nm.1 == 0
                 then Ret (inl nm.2)
                 else if 100 < nm.1
                      then Ret (inr (nm.1.-1, nm.2 - 10))
                      else Ret (inr (nm.1.+1, nm.2 + 11))) (n.+1, m)
《 fixpointE 》
= (if 100 < m
   then Ret (inr (n, m - 10))
   else Ret (inr (n.+2, m + 11))) >>= sum_rect Ret (while mc91_body)
《 m < 101 》
= Ret (inr (n.+2, m + 11)) >>= sum_rect Ret (while mc91_body)
```

Proof using equational reasoning

- We proved $89 < m < 101 \Rightarrow mc91\ (m) = mc91\ (m + 1)$.

`bindretf`	`Ret a >>= f = f a`
`fixpointE`	`while f a = (f a) >>= (sum_rect Ret (while f))`

```
mc91 n m
= 《 definition of mc91 》
while (fun nm => if nm.1 == 0
                 then Ret (inl nm.2)
                 else if 100 < nm.1
                      then Ret (inr (nm.1.-1, nm.2 - 10))
                      else Ret (inr (nm.1.+1, nm.2 + 11))) (n.+1, m)
《 fixpointE 》
= (if 100 < m
   then Ret (inr (n, m - 10))
   else Ret (inr (n.+2, m + 11))) >>= sum_rect Ret (while mc91_body)
《 m < 101 》
= Ret (inr (n.+2, m + 11)) >>= sum_rect Ret (while mc91_body)
《 bindretf 》
= while mc91_body (n.+2, m + 1)
```

Proof using equational reasoning

- We proved $89 < m < 101 \Rightarrow mc91\ (m) = mc91\ (m+1)$.

`bindretf`	`Ret a >>= f = f a`
`fixpointE`	`while f a = (f a) >>= (sum_rect Ret (while f))`

```
mc91 n m
= 《 definition of mc91 》
while (fun nm => if nm.1 == 0
                 then Ret (inl nm.2)
                 else if 100 < nm.1
                      then Ret (inr (nm.1.-1, nm.2 - 10))
                      else Ret (inr (nm.1.+1, nm.2 + 11))) (n.+1, m)
《 fixpointE 》
= (if 100 < m
   then Ret (inr (n, m - 10))
   else Ret (inr (n.+2, m + 11))) >>= sum_rect Ret (while mc91_body)
《 m < 101 》
= Ret (inr (n.+2, m + 11)) >>= sum_rect Ret (while mc91_body)
《 bindretf 》
= while mc91_body (n.+2, m + 1)
 ...
= while mc91_body (n.+1, m + 11 - 10) = mc91 n (m+1)
```

Model: delay monad [Cap05] and `wBisim`

- The delay monad is an instance of the complete Elgot monad.
- `Delay` : (`A` : Type) → (maximum fixpoint of `X = A + X`).

```
CoInductive Delay (A : Type) : Type :=
  | DNow : A -> Delay A
  | DLater : Delay A -> Delay A. (* one step of computation *)
```

- `DLater` expresses a step of computation.
- `wBisim` (≈) is expressing computational equivalence.

$$\mathtt{d1} \approx \mathtt{d2}$$

$$\Updownarrow$$

(1) `d1` and `d2` are equal ignoring finitely many applications of `DLater`

or (2) both `d1` and `d2` are the result of infinitely many applications of `DLater`

Monad structure of the Delay monad.

- Monad operators.

```
Let ret (a : A) := DNow a.
CoFixpoint bind (m : Delay A) (f : A -> Delay B) :=
  match m with
  | DNow a => f a
  | DLater d => DLater (bind d f)
  end.
```

Definition of `while` for the Delay monad

```
CoFixpoint while (body : A -> M (B + A)) : A -> M B :=
  fun a => (body a >>=
           (fun ab => match ab with
                      | inr a => DLater (while body a)
                      | inl b => DNow b end)).
```

- `fixpointE`: loop unrolling.

Case : `body x = ret (inr y)`

```
while body a  ≈  (body a) >>= (sum_rect ret (while body))
```

```
while body x
= body x >>= (fun ab => match ab with
                 inr a => DLater (while body a)
                 inl b => DNow b end).
= (* body x = ret (inr y) *) DLater (while body y)
≈ while body y
= (* body x = ret (inr y) *) (body x) >>= (sum_rect ret (while f))
```

Combination of references with `while` statements

```
let fact n =
let r = ref 1 in
let l = ref 1 in



while !l <= n do

  r := !r * !l;
  l := !l + 1;

done;
!r
```

⇒

```
Definition factdts n :=
do r <- cnew ml_int 1;
do l <- cnew ml_int 1;
do _ <-
while (fun (_ : unit) =>
        do i <- cget l;
        if i <= n
        then do v <- cget r;
             do _ <- cput r (i * v);
             do _ <- cput l (i.+1);
             Ret (inr tt)
        else Ret (inl tt)) tt;
do v <- cget r; Ret v.
```

- We combine other computational effects using a monad transformer.

Combination with other effects using a monad transofrmer

- Monae already supports monad transformers [AN20]

Typed-store monad transformer

- The typed-store monad is introduced for expressing OCaml references. [AGS25].
- This monad has cnew, cget and cput operators.
- This is defined as the composition of `MS` (state) and `MX` (exception).

```
Definition MTS : monad -> monad :=
  (MS (seq binding)) \o (MX unit).
```

`MS` preserves the complete Elgot monad structure.
`MX` preserves the complete Elgot monad structure .

$\Downarrow$

`MTS` preserves the complete Elgot monad structure.

Conclusion

- Monadic equational reasoning for programs containing `while` statements.
  - The interface is based on the complete Elgot monad.
  - A Delay monad proves the soundness.
  - Monae hides coinductive definitions.
- Computational effects are combined using monad transformers.
  - We show the state monad transformer `MS` and the exception monad transformer `MX` preserve the structure of complete Elgot monad.
- Examples (collatz, McCarthy's 91, fafctorial with a while loops).
- Future work
  - Definition of a generalized fixpoint operator.
  - Verification of partial correctness.

[AGS25] Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa.
A practical formalization of monadic equational reasoning in dependent-type theory.
Journal of Functional Programming, 35:e1, 2025.

[AMV10] Jirí Adámek, Stefan Milius, and Jirí Velebil.
Equational properties of iterative monads.
Information and Computation, 208(12):1306–1348, 2010.

[AN20] Reynald Affeldt and David Nowak.
Extending equational monadic reasoning with monad transformers.
In *26th International Conference on Types for Proofs and Programs (TYPES 2020), March 2–5, 2020, University of Turin, Italy*, volume 188 of *LIPIcs*, pages 2:1–2:21, 2020.

Reference ii

[ANS19] Reynald Affeldt, David Nowak, and Takafumi Saikawa.
A hierarchy of monadic effects for program verification using equational reasoning.
In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019*, volume 11825 of *LNCS*, pages 226–254. Springer, 2019.

[Cap05] Venanzio Capretta.
General recursion via coinductive types.
Logical Methods in Computer Science, 1(2), 2005.

[GH11] Jeremy Gibbons and Ralf Hinze.
Just do it: simple monadic equational reasoning.
In *16th ACM SIGPLAN international conference on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011*, pages 2–14. ACM, 2011.