Extending Monae to formalize quicksort using monads in Coq

定理証明支援系Coqでのモナドを用いたクイックソートの形式化とMonaeの拡張

Ayumu Saito (Tokyo Institute of Technology)

Reynald Affeldt (AIST)

Background

- **Functional programming language are suitable for equational reasoning**
  - Referential transparency
- **Many programs have side effects**
  - They can be dealt with monads
- **A lot of interest in formal verification with monadic effects**
  - [Gibbons and Hinze, ICFP 2011]
  - [Mu and Chiang, FLOPS 2020]
  - [Affeldt, Garrigue, Nowak, Saikawa, JFP 2021]

Purpose and Problem

- **Purpose: Support the formal verification of programs using monads**
- **Problem: Develop a usable framework is technically a challenging task**
  - Monadic effect is the result of the combination of several interfaces of monad
  - A type-based proof assistant (Coq, Agda) requires termination proofs for every function
  - Pre-existing libraries of lemmas are needed for large experiments

Contributions

- **Improve an existing formalization of monadic equational reasoning**
  - **Extend** Monae to deal with plusMonad and its combination with arrayMonad
  - **Explain** how to deal with non-structural recursive functions
  - **Enrich** the support libraries of Monae
    - With refinement for specification
    - With nondeterministic permutations for experiments
- **Application:**
  - Without any axiomatized facts, we reproduce the proofs by Mu and Chiang [FLOPS 2020]

Outline

- **Extend Monae**
- **Explain how to deal with non-structural recursive functions**
- **Enrich libraries**
- **Quicksort experiments**
- **Conclusion**

Functor using Hierarchy-Builder[Cohen, Tassi, Sakaguchi, FSCD 2020]

Goal: arrayMonad

- **Hierarchy-Builder is to build hierarchies of mathematical structures**

```
HB.mixin Record isFunctor (M : Type -> Type) := {
  actm : forall A B : Type, (A -> B) -> M A -> M B ;
  functor_id : FunctorLaws.id actm ;
  functor_o : FunctorLaws.comp actm }.
```

- `Type` ≈ Set category
- `M` ≈ Action on objects
- `actm` ≈ Action on morphisms

- **Functor Laws**
  - Functors preserve identity morphisms (`functor_id`)

$$actm\ id = id$$

  - Functors preserve composition of morphisms (`functor_o`)

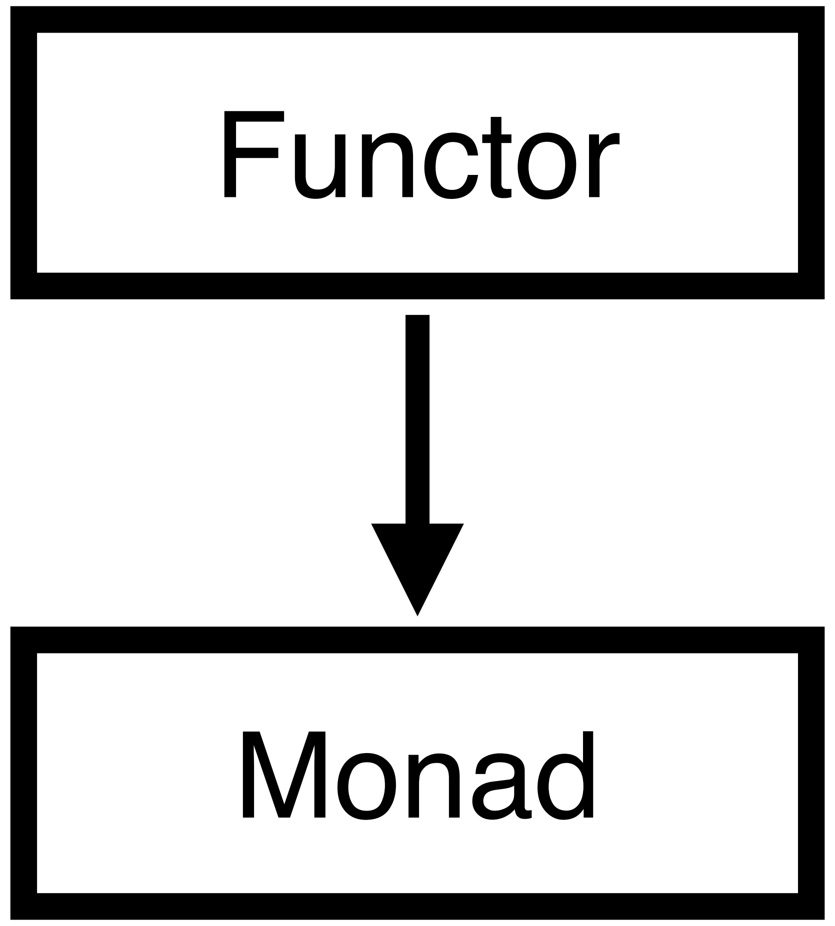
$$actm\ f \circ g = actm\ f \circ actm\ g$$

Monad using Hierarchy-Builder

- **The mixin extends the structure functor**

```
HB.mixin Record isMonad (M : Type -> Type) of Functor M := {
  ret : idfun ~> M ;
  join : M \o M ~> M ;
  joinretM : JoinLaws.left_unit ret join ;
  joinMret : JoinLaws.right_unit ret join ;
  joinA : JoinLaws.associativity join }.
```

- `M` ≈ Functor
- `~>` ≈ Natural translation
- `idfun` ≈ Identity functor
- `\o` ≈ composition

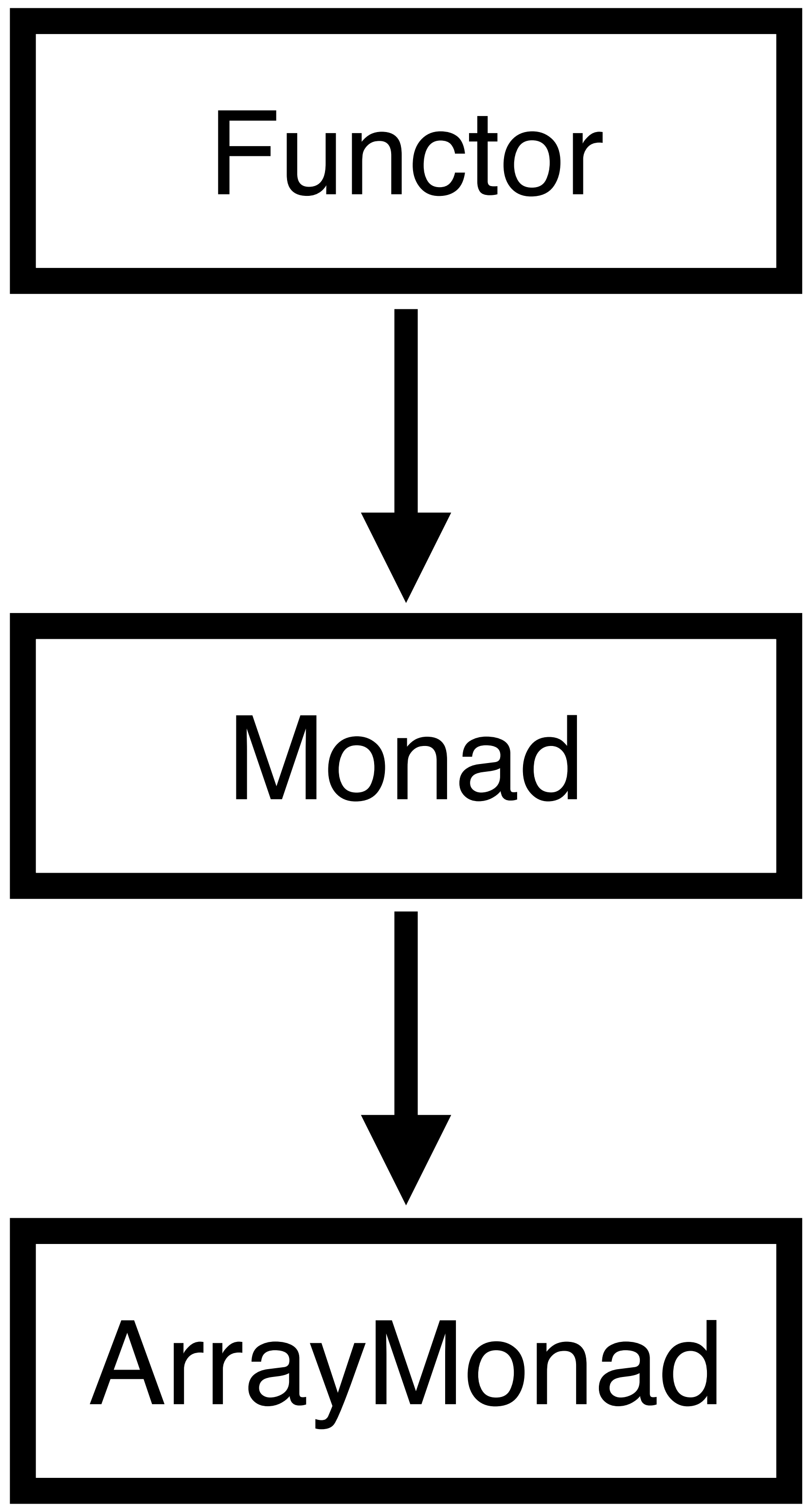


- **Monad Laws**
  - `joinretM` $join \circ ret = id$
  - `joinMret` $join \circ actm\ ret = id$
  - `joinA` $join \circ actm\ join = join \circ join$

ArrayMonad using Hierarchy-Builder

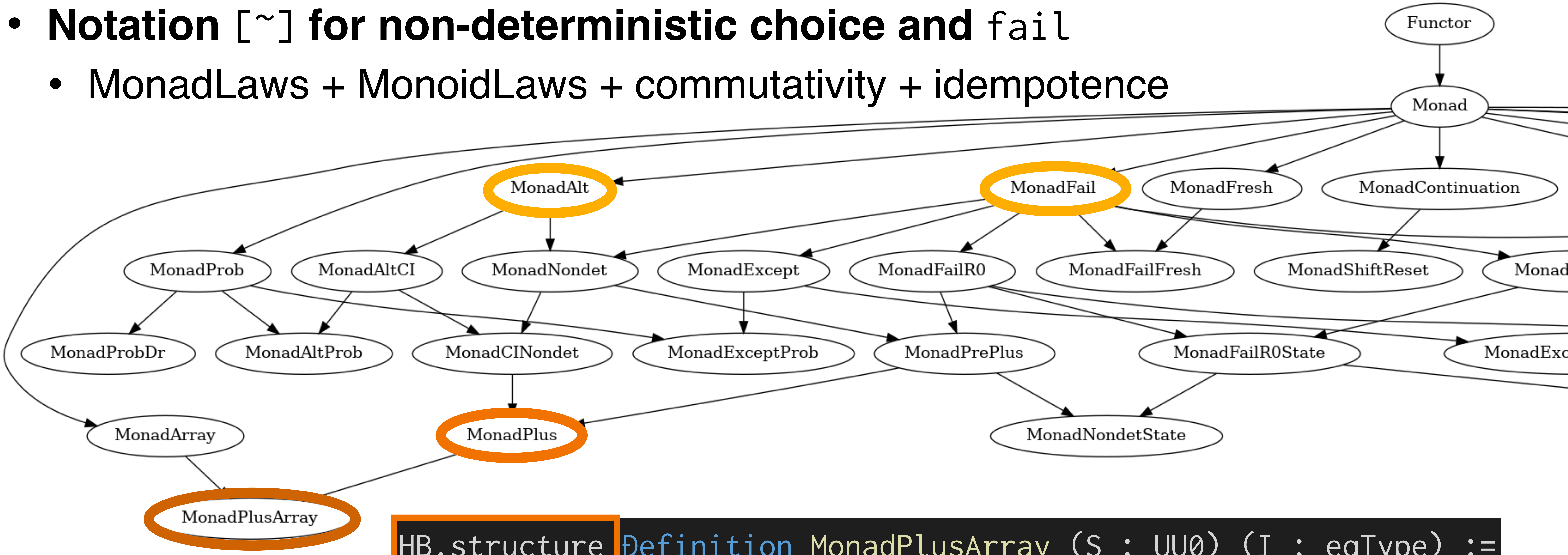
```
HB.mixin Record isMonadArray (S : Type) (I : eqType) (M : Type -> Type) of Monad M := {
  aget : I -> M S ;
  aput : I -> S -> M unit ;
  aputput : forall i s s', aput i s >> aput i s' = aput i s' ;
  aputget : forall i s (A : Type) (k : S -> M A),
    aput i s >> aget i >>= k = aput i s >> k s ;
  agetpustskip : (省略) ;
  agetget : (省略) ;
  agetC : (省略) ;
  aputC : (省略) ;
  aputgetC : (省略) ;
}.
```

- aget : get from index i
- aput : put to index i



plusMonad and plusArrayMonad

- **Notation `[~]` for non-deterministic choice and `fail`**
  - MonadLaws + MonoidLaws + commutativity + idempotence




```
HB.structure Definition MonadPlusArray (S : UU0) (I : eqType) :=
  { M of MonadPlus M & isMonadArray S I M}.
```

Outline

- Extend Monae
- Explain how to deal with non-structural recursive functions
- Enrich libraries
- Quicksort experiments
- Conclusion

Difficulties with non-structurally recursive functions

`Function` approach

- **We cannot directly use standard coq tools such as `Fixpoint`**
- **`qperm` returns permutations non-deterministically**
  - Use intermediate function `splits`:

```
Fixpoint splits {M : plusMonad} A (s : seq A) : M (seq A * seq A) :=
  if s isn't x :: xs then Ret ([::], [::]) else
    splits xs >>= (fun yz => Ret (x :: yz.1, yz.2) [~] Ret (yz.1, x :: yz.2)).
```

  - Ill-typed error

```
Fail Function qperm (s : seq A) {measure size s} : M (seq A) :=
    if s isn't x :: xs then Ret [::] else
    splits xs >>= (fun '(ys, zs) =>
      liftM2 (fun a b => a ++ x :: b) (qperm ys) (qperm zs)).
```

Difficulties with non-structurally recursive functions

`Program Definition/Fix` approach

- **We cannot directly use even more primitive commands like Program Definition / Fix**

  1. Program Definition

```
Program Definition qperm' (s : seq A)
    (f : forall s', size s' < size s -> M (seq A)) : M (seq A) :=
  if s isn't x :: xs then Ret [::] else
    splits xs >>= (fun '(ys, zs) =>
    liftM2 (fun a b => a ++ x :: b) (f ys _) (f zs _)).
```

This fails because we lose size information about return value of splits

  2. Fix (from the standard library)

```
Definition qperm : seq A -> M (seq A) :=
  Fix (@well_founded_size _) (fun _ => M _) qperm'.
```

Add dependent types to intermediate functions

Approach 1: rewrite the intermediate function

- **tsplits returns ‘bounded sequences’ (.-bseq notation)**

```
Fixpoint tsplits {M : plusMonad} A (s : seq A)
    : M ((size s).-bseq A * (size s).-bseq A) :=
  if s isn't x :: xs then Ret ([bseq of [::]], [bseq of [::]])
  else tsplits xs >>= (fun '(ys, zs) =>
    Ret ([bseq of x :: ys], widen_bseq (leqnSn _) zs) [~]
    Ret (widen_bseq (leqnSn _) ys, [bseq of x :: zs])).
```

- **Using `tsplits`, we can complete the definition of `qperm`**

```
Program Definition qperm' (s : seq A)
    (f : forall s', size s' < size s -> M (seq A)) : M (seq A) :=
  if s isn't x :: xs then Ret [::] else
    tsplits xs >>= (fun '(ys, zs) =>
    liftM2 (fun a b => a ++ x :: b) (f ys _) (f zs _)).
```

Add dependent types to intermediate functions

Approach 2: compose with a dependent assertion

- **We introduce a dependently-typed ‘assertion’**

```
Definition dassert (p : pred A) (a : A) : M { a | p a } :=
  if Bool.bool_dec (p a) true is left H then Ret (exist p a H) else fail.
```

- **Example: consider ‘ipartl' that returns a pair of natural numbers**
  We augment its type with proof that the return value is smaller than the inputs

```
Definition dipartl (p : E) (i : Z) (y z x : nat) :
  M {n | (n.1 <= x + y + z) && (n.2 <= x + y + z) } :=
  ipartl p i y z x >>=
    (dassert [pred n | (n.1 <= x + y + z) && (n.2 <= x + y + z)]).
```

Outline

- Extend Monae
- Explain how to deal with non-structural recursive functions
- Enrich libraries
- Quicksort experiments
- Conclusion

New libraries

Refinement [Mu and Chiang, FLOPS 2020]

- **`m1` refines `m2` when every result of m1 is a result of m2**

```
Definition refin (M : altMonad) A (m1 m2 : M A) : Prop := m1 [~] m2 = m2.

Notation "m1 `<=` m2" := (refin m1 m2) : monae_scope.
```

- **Notation for pointwise-lifting**

```
Definition lrefin (M : altMonad) A B (f g : A -> M B) := forall x, f x `<=` g x.

Notation "f `<.=` g" := (lrefin f g) : monae_scope.
```

New libraries

Difficulties with nondeterministic permutations

- **qperm is useful to write specification, but difficult to use**
  - More than 5 axioms on Mu and Chiang’s proof [FLOPS 2020]
  - Example (Agda code):

```
postulate
  return⊑perm : {{_ : MonadPlus M}} → return {M} {List A} ⊑ perm
  perm-idempotent : {{_ : MonadPlus M}} (xs : List A)
                  → perm xs >>= perm ≈ perm xs
  perm-snoc : {{_ : MonadPlus M}} (xs : List A) (x : A)
            → perm (xs ++ [ x ]) ≈ perm xs >>= λ xs' -> perm (xs' ++ [ x ])

postulate
  perm-preserves-length : {{_ : MonadPlus M}} {{_ : Ord A}} → perm preserves (length {_} {A})
  perm-preserves-all : {{_ : MonadPlus M}} {{_ : Ord A}}
                     → (p : A → Bool) → perm preserves (all p)
```

Proof-friendly definition of nondeterministic permutations

- **Idea: use a simpler definition with Fixpoint only**

```
Fixpoint insert (a : A) (s : seq A) : M (seq A) :=
  if s isn't h :: t then Ret [:: a] else
  Ret (a :: h :: t) [~] fmap (cons h) (insert a t).

Fixpoint perm (s : seq A) : M (seq A) :=
  if s isn't h :: t then Ret [::] else perm t >>= insert h.
```

1. Prove the equivalence of `perm` and `qperm`
2. Transport the properties of `perm` to `qperm`

Outline

- Extend Monae
- Explain how to deal with non-structural recursive functions
- Enrich libraries
- **Quicksort experiments**
- Conclusion

Quicksort on lists (no arrayMonad)

- **Quicksort on lists in Coq**

```
Function qsort (s : seq T) {measure size s} : seq T :=
  if s isn't h :: t then [::]
  else let: (ys, zs) := partition h t in
       qsort ys ++ h :: qsort zs.
```

- **Obviously correct sorting algorithm**

```
Definition slowsort : seq T -> M (seq T) := qperm >=> assert sorted.
```

- **Specification of quicksort**

```
Lemma qsort_slowsort : Ret \o qsort `<.=` slowsort.
```

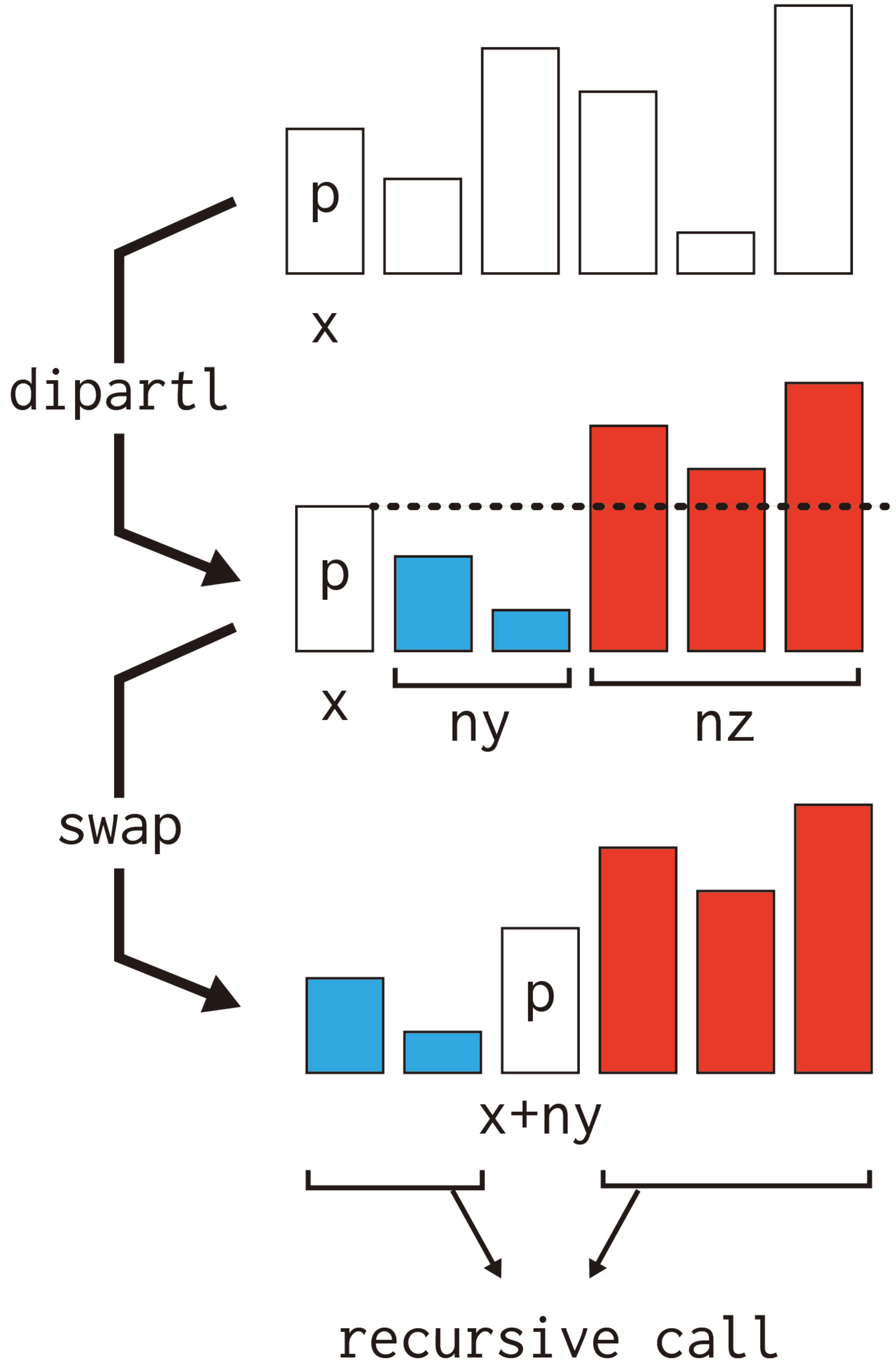
Quicksort with arrayMonad in Coq

```
Program Fixpoint iqsort' xn (f : forall ym, ym.2 < xn.2 -> M unit) : M unit :=
  match xn.2 with
  ¦ 0 => Ret tt
  ¦ n.+1 => aget xn.1 >>= (fun p =>
              dipartl p (xn.1 + 1) 0 0 n >>= (fun nynz =>
                let ny := nynz.1 in
                let nz := nynz.2 in
                swap xn.1 (xn.1 + ny) >>
                f (xn.1, ny) _ >> f (xn.1 + ny + 1, nz) _))
  end.
```

- **Specification of quicksort using arrayMonad and refinement**

```
Lemma iqsort_slowsort x (s : seq E) :
  writeList x s >> iqsort (x, size s) `<=` slowsort s >>= writeList x.
```

```
Program Fixpoint iqsort' xn (f : forall ym, ym.2 < xn.2 -> M unit) : M unit :=
  match xn.2 with
  | 0 => Ret tt
  | n.+1 => aget xn.1 >>= (fun p =>
              dipartl p (xn.1 + 1) 0 0 n >>= (fun nynz =>
                let ny := nynz.1 in
                let nz := nynz.2 in
                swap xn.1 (xn.1 + ny) >>
                f (xn.1, ny) _ >> f (xn.1 + ny + 1, nz) _))
  end.
```



Outline

- Extend Monae
- Explain how to deal with non-structural recursive functions
- Enrich libraries
- Quicksort experiments
- Conclusion

Conclusion

- **We extended monae with plusArrayMonad and libraries about nondeterministic permutations and sorting**
  - plusArrayMonad using Hierarchy-Builder
- **As an application, we could formalize [Mu and Chiang, FLOPS 2020] without axioms**

- **Future work:**
  - Formalization of “Handling Local State with Global State” [Pauwels, Schrijvers, Mu, MPC 2019]