Composing bidirectional programs monadically

Li-yao Xia,[1] Dominic Orchard,[2] Meng Wang,[3]

[1]University of Pennsylvania
[2]University of Kent [3]University of Bristol

ESOP 2019, April 8

Composing bidirectional programs monadically

Pairs of programs in "opposite directions".

Composing bidirectional programs monadically

Pairs of programs in “opposite directions”.

- Parsers - printers

```
String                AST
\x -> x     parse->   Fun "x" (Var "x")
            <-print
```

Composing bidirectional programs monadically

Pairs of programs in "opposite directions".

- Parsers - printers

```
String                    AST
\x -> x     parse->   Fun "x" (Var "x")
            <-print
```

- Lenses (getters - setters)

```
Source (Database)                    View (Row)
```

Composing bidirectional programs monadically

Pairs of programs in “opposite directions”.

- Parsers - printers

```
String                AST
\x -> x    parse->    Fun "x" (Var "x")
           <-print
```

- Lenses (getters - setters)

```
Source (Database)                      View (Row)
(Alita, 220) ; (Sechs, 2)    get->    (Alita, 220)
```

Composing bidirectional programs monadically

Pairs of programs in "opposite directions".

- Parsers - printers

```
String                 AST
\x -> x     parse->    Fun "x" (Var "x")
            <-print
```

- Lenses (getters - setters)

```
Source (Database)                    View (Row)
(Alita, 220) ; (Sechs, 2)   get->    (Alita, 220)
                                           ↓
(Alita, 221) ; (Sechs, 2)   <-set    (Alita, 221)
```

Composing bidirectional programs monadically

Pairs of programs in "opposite directions".

- Parsers - printers

```
String                  AST
\x -> x     parse->     Fun "x" (Var "x")
            <-print
```

- Lenses (getters - setters)

```
Source (Database)                      View (Row)
(Alita, 220) ; (Sechs, 2)    get->     (Alita, 220)
                                             ↓
(Alita, 221) ; (Sechs, 2)    <-set     (Alita, 221)
```

Subject to "*round-tripping laws*".

Composing bidirectional programs monadically

Pairs of programs in "opposite directions".

- Random generators - predicates

```
randomSortedList :: Prob [Int]
isSortedList :: [Int] -> Bool
```

Composing bidirectional programs monadically

Pairs of programs in "opposite directions".

- Random generators - predicates

```
randomSortedList :: Prob [Int]
isSortedList :: [Int] -> Bool
```

- For random testing of invariants

```
dropSorted :: Property      -- Using QuickCheck
dropSorted =
  forAll randomSortedList (\ (xs :: [Int]) ->
    isSortedList (drop 1 xs)))
```

Composing bidirectional programs monadically

Pairs of programs in "opposite directions".

- Random generators - predicates

```
randomSortedList :: Prob [Int]
isSortedList :: [Int] -> Bool
```

- For random testing of invariants

```
dropSorted :: Property      -- Using QuickCheck
dropSorted =
  forAll randomSortedList (\ (xs :: [Int]) ->
    isSortedList (drop 1 xs)))
```

- "Round-trip":

$$\mathbb{P}\big(\texttt{randomSortedList = [1,2,3]}\big) > 0$$

$$\texttt{isSortedList [1,2,3] = True}$$

- General idea: same relation viewed in two directions.

Composing bidirectional programs monadically

- General idea: same relation viewed in two directions.

Basic plan of a talk on bidirectional programming:

1. How to obtain both directions from a single description?

Composing bidirectional programs monadically

- General idea: same relation viewed in two directions.

Basic plan of a talk on bidirectional programming:

1. How to obtain both directions from a single description?
2. What *round-tripping* guarantees to expect?

Composing **bidirectional programs** monadically

- General idea: same relation viewed in two directions.

Basic plan of a talk on bidirectional programming:

1. How to obtain both directions from a single description?
2. What *round-tripping* guarantees to expect?
   - `forall v. parse (print v) = v`
   - Lens laws: `get (set s v) = v`
   - Soundness/completeness of generators

Composing bidirectional programs monadically

- General idea: same relation viewed in two directions.

Basic plan of a talk on bidirectional programming:

1. How to obtain both directions from a single description?
2. What *round-tripping* guarantees to expect?
   - `forall v. parse (print v) = v`
   - Lens laws: `get (set s v) = v`
   - Soundness/completeness of generators

Running example for this talk: parsers - printers.

- Present approach: **combinators** to compose bidirectional programs. Typical features:

Composing bidirectional programs monadically

- Present approach: **combinators** to compose bidirectional programs. Typical features:
  - DSL as library (= EDSL).

Composing bidirectional programs monadically

- Present approach: **combinators** to compose bidirectional programs. Typical features:
  - DSL as library (= EDSL).
  - Fitting within host language poses design challenges.

Composing bidirectional programs monadically

- Present approach: **combinators** to compose bidirectional programs. Typical features:
  - DSL as library (= EDSL).
  - Fitting within host language poses design challenges.
  - "Round-tripping" properties usually preserved by combinators (compositionality). A more complicated story here.

Composing bidirectional programs monadically

- Present approach: **combinators** to compose bidirectional programs. Typical features:
  - DSL as library (= EDSL).
  - Fitting within host language poses design challenges.
  - "Round-tripping" properties usually preserved by combinators (compositionality). A more complicated story here.
- What combinators to choose?

Composing bidirectional programs monadically

- Present approach: **combinators** to compose bidirectional programs. Typical features:
  - DSL as library ($=$ EDSL).
  - Fitting within host language poses design challenges.
  - "Round-tripping" properties usually preserved by combinators (compositionality). A more complicated story here.
- What combinators to choose?
  - We can try to adapt known abstractions.

Monads

- A general interface to compose programs.

```
-- M :: Type -> Type

(>>=) :: M a -> (a -> M b) -> M b
return :: a -> M a
-- + monad laws
```

Monads

- A general interface to compose programs.

```
-- M :: Type -> Type

(>>=) :: M a -> (a -> M b) -> M b
return :: a -> M a
-- + monad laws
```

- Example: monadic parser (`M = Parser`).

```
parseString :: Parser String  -- String = [Char]
parseString =
  parseInt >>= (\ (n :: Int) ->
    replicateM n parseChar)

parseInt  :: Parser Int
parseChar :: Parser Char
replicateM :: Int -> Parser a -> Parser [a]
```

Monads, an unlikely candidate

- Monads (e.g., parsers) are *covariant* functors.

```
type Parser a = [Char] -> (a, [Char])
fmap :: (a -> b) -> Parser a -> Parser b
-- Definable from (>>=) and return
```

Monads, an unlikely candidate

- Monads (e.g., parsers) are *covariant* functors.

```
type Parser a = [Char] -> (a, [Char])
fmap :: (a -> b) -> Parser a -> Parser b
-- Definable from (>>=) and return
```

- Printers are *contravariant*.

```
type Printer a = a -> [Char]
comap :: (b -> a) -> Printer a -> Printer b
-- can be defined
```

Can a type be both covariant and contravariant?

```
fmap  :: (a -> b) -> M a -> M b
comap :: (b -> a) -> M a -> M b
```

Can a type be both covariant and contravariant?

```
fmap  :: (a -> b) -> M a -> M b
comap :: (b -> a) -> M a -> M b
```

1. No, it would be *phantom*: the definition of `(M a)` couldn't use `a`.

Can a type be both covariant and contravariant?

```
fmap  :: (a -> b) -> M a -> M b
comap :: (b -> a) -> M a -> M b
```

1. No, it would be *phantom*: the definition of `(M a)` couldn't use `a`.
2. No, but we can use an *invariant type* instead.
   ```
   type (a <-> b) = (a -> b, b -> a)
   invmap :: (a <-> b) -> M a -> M b
   ```
   (Popular approach in related work.)

Can a type be both covariant and contravariant?

```
fmap  :: (a -> b) -> M a -> M b
comap :: (b -> a) -> M a -> M b
```

1. No, it would be *phantom*: the definition of `(M a)` couldn't use `a`.
2. No, but we can use an *invariant type* instead.
   ```
   type (a <-> b) = (a -> b, b -> a)
   invmap :: (a <-> b) -> M a -> M b
   ```
   (Popular approach in related work.)
3. **Yes**, with a twist: *profunctors*.

Profunctors

- Covariance, contravariance, pick two.

Profunctors

- Covariance, contravariance, pick two.

```
-- P :: Type -> Type -> Type
```

Profunctors

- Covariance, contravariance, pick two.

```
-- P :: Type -> Type -> Type

fmap  :: (a -> b) -> P x a -> P x b
```

Profunctors

- Covariance, contravariance, pick two.

```
-- P :: Type -> Type -> Type

fmap  :: (a -> b) -> P x a -> P x b

comap :: (b -> a) -> P a y -> P b y
```

Monadic profunctors

- Covariance, contravariance, pick two.

```
-- P :: Type -> Type -> Type

fmap  :: (a -> b) -> P x a -> P x b

comap :: (b -> a) -> P a y -> P b y
```

- **New mix:** for any `x`, `(P x)` is a monad

```
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a
```

Monadic profunctors

- Covariance, contravariance, pick two.

```haskell
-- P :: Type -> Type -> Type

fmap  :: (a -> b) -> P x a -> P x b

comap :: (b -> a) -> P a y -> P b y
```

- **New mix:** for any `x`, `(P x)` is a monad

```haskell
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a

-- Definable from (>>=) and return
fmap :: (a -> b) -> P x a -> P x b
```

Monadic profunctors

- In summary, minimal definition:

```
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a
-- forall x. Monad (P x)

comap :: (b -> a) -> P a y -> P b y
```

Monadic profunctors

- In summary, minimal definition:

```
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a
-- forall x. Monad (P x)

comap :: (b -> a) -> P a y -> P b y
```

- Take `Monad`, add one more type parameter and one more function, that's all we need for bidirectional programming.

Monadic profunctors

- In summary, minimal definition:

```
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a
-- forall x. Monad (P x)

comap :: (b -> a) -> P a y -> P b y
```

- Take `Monad`, add one more type parameter and one more function, that's all we need for bidirectional programming.
- This work: study properties of this simple interface.

Parser monad (again)

```haskell
parseString :: Parser String
parseString =
  parseInt >>= (\n ->
    replicateM n parseChar)

-- assuming
parseInt  :: Parser Int
parseChar :: Parser Char
replicateM :: Int -> Parser a -> Parser [a]
```

Bidirectional parser profunctor monad

```haskell
biparseString :: Biparser String String
biparseString =
  comap length biparseInt >>= (\n ->
    replicateP n biparseChar)

-- assuming
biparseInt  :: Biparser Int Int
biparseChar :: Biparser Char Char
replicateP :: Int -> Biparser x a -> Biparser [x] [a]
```

Bidirectional parser profunctor monad

```
biparseString :: Biparser String String
biparseString =
  comap length biparseInt >>= (\n ->
    replicateP n biparseChar)

-- assuming
biparseInt  :: Biparser Int Int
biparseChar :: Biparser Char Char
replicateP :: Int -> Biparser x a -> Biparser [x] [a]
```

- Both a parser and a printer.

Bidirectional parser profunctor monad

```
-- P :: Type -> Type -> Type

comap :: (y -> x) -> P x a -> P y a
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a
-- i.e., forall x. Monad (P x)
```

- Three monadic profunctors:

Bidirectional parser profunctor monad

```
-- P :: Type -> Type -> Type

comap :: (y -> x) -> P x a -> P y a
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a
-- i.e., forall x. Monad (P x)
```

- Three monadic profunctors:

```
type Parser' x a = ([Char] -> (a, [Char]))  -- Parser a
```

Bidirectional parser profunctor monad

```
-- P :: Type -> Type -> Type

comap :: (y -> x) -> P x a -> P y a
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a
-- i.e., forall x. Monad (P x)
```

- Three monadic profunctors:

```
type Parser' x a = ([Char] -> (a, [Char]))  -- Parser a
```

```
type Printer x a = (x -> ([Char], a))
```

Bidirectional parser profunctor monad

```
-- P :: Type -> Type -> Type

comap :: (y -> x) -> P x a -> P y a
(>>=) :: P x a -> (a -> P x b) -> P x b
return :: a -> P x a
-- i.e., forall x. Monad (P x)
```

- Three monadic profunctors:

```
type Parser' x a = ([Char] -> (a, [Char]))  -- Parser a

type Printer x a = (x -> ([Char], a))

type Biparser x a = (Parser' x a, Printer x a)
-- Parser-printer pairs
```

A concrete example in detail

```
comap length biparseInt :: Biparser [Char] Int
```

A concrete example in detail

```
comap length biparseInt :: Biparser [Char] Int
```

- As a parser:

  ```
  parseInt :: Parser Int  -- comap is erased
  ```

A concrete example in detail

```
comap length biparseInt :: Biparser [Char] Int
```

- As a parser:

```
parseInt :: Parser Int  -- comap is erased
```

- As a printer:

```
(\  s       -> let n = length s in
               (printInt n,   n))
 :: [Char] -> ([Char]     , Int)
 -- Printer [Char] Int
 --          ^       ^ result, printed value
 --          ^ "context" around value to print

 -- given
 printInt :: Int -> [Char]
```

A general recipe of profunctor monads

- "Forward" and "backward" profunctor monads.

```
type Fwd m x a = m a
```

A general recipe of profunctor monads

- "Forward" and "backward" profunctor monads.

```
type Fwd m x a = m a
type Parser' x a = [Char] -> (a, [Char])    --
type Parser' x a = Fwd (State [Char]) x a  -- same
```

A general recipe of profunctor monads

- "Forward" and "backward" profunctor monads.

```
type Fwd m x a = m a
type Parser' x a = [Char] -> (a, [Char])    --
type Parser' x a = Fwd (State [Char]) x a  -- same

type Bwd n x a = x -> n a
```

A general recipe of profunctor monads

- "Forward" and "backward" profunctor monads.

```
type Fwd m x a = m a
type Parser' x a = [Char] -> (a, [Char])    --
type Parser' x a = Fwd (State [Char]) x a  -- same

type Bwd n x a = x -> n a
type Printer x a = x -> ([Char], a)          --
type Printer x a = Bwd (Writer [Char]) x a  -- same
```

A general recipe of profunctor monads

- "Forward" and "backward" profunctor monads.

```
type Fwd m x a = m a
type Parser' x a = [Char] -> (a, [Char])    --
type Parser' x a = Fwd (State [Char]) x a  -- same

type Bwd n x a = x -> n a
type Printer x a = x -> ([Char], a)          --
type Printer x a = Bwd (Writer [Char]) x a  -- same

type (p :*: q) x a = (p x a, q x a)
type Biparser x a = (Parser' :*: Printer) x a
```

A general recipe of profunctor monads

- "Forward" and "backward" profunctor monads.

```
type Fwd m x a = m a
type Parser' x a = [Char] -> (a, [Char])    --
type Parser' x a = Fwd (State [Char]) x a   -- same

type Bwd n x a = x -> n a
type Printer x a = x -> ([Char], a)          --
type Printer x a = Bwd (Writer [Char]) x a   -- same

type (p :*: q) x a = (p x a, q x a)
type Biparser x a = (Parser' :*: Printer) x a
```

- What relation between `m` in `Fwd m` and `n` in `Bwd n`? (unsolved)

Round-tripping properties

```
parse :: Biparser a a -> [Char] -> Maybe a
print :: Biparser a a -> a -> [Char]
```

Round-tripping properties

```
parse :: Biparser a a -> [Char] -> Maybe a
print :: Biparser a a -> a -> [Char]
```

- `p :: Biparser a a` is *forward round-tripping* if

  ```
  parse p s = Just a   ->   print p a = s
  ```

Round-tripping properties

```
parse :: Biparser a a -> [Char] -> Maybe a
print :: Biparser a a -> a -> [Char]
```

- `p :: Biparser a a` is *forward round-tripping* if

  ```
  parse p s = Just a   ->   print p a = s
  ```

- `p :: Biparser a a` is *backward round-tripping* if

  ```
  print p a = s   ->   parse p s = Just a
  parse p (print p a) = Just a   -- equivalently
  ```

Round-tripping properties

```
parse :: Biparser a a -> [Char] -> Maybe a
print :: Biparser a a -> a -> [Char]
```

- `p :: Biparser a a` is *forward round-tripping* if
  ```
  parse p s = Just a   ->   print p a = s
  ```
- `p :: Biparser a a` is *backward round-tripping* if
  ```
  print p a = s   ->   parse p s = Just a
  parse p (print p a) = Just a    -- equivalently
  ```
- Sadly, round-tripping (bwd or fwd) is **not** guaranteed by construction!
  ```
  comap :: (y -> x) -> P x a -> P y a
  (>>=) :: P x a -> (a -> P x b) -> P x b
  ```

Verifying round-tripping properties

Verifying round-tripping properties

- Baseline, naive verification method: extract parser, extract printer, check that they match.

Verifying round-tripping properties

- Baseline, naive verification method: extract parser, extract printer, check that they match.
- Can we do better by exploiting the **shared structure** of biparsers?

Verifying round-tripping properties

- Baseline, naive verification method: extract parser, extract printer, check that they match.
- Can we do better by exploiting the **shared structure** of biparsers?
- Plan:
  1. *Weaken* round-tripping to be compositional (i.e., property guaranteed by construction).

Verifying round-tripping properties

- Baseline, naive verification method: extract parser, extract printer, check that they match.
- Can we do better by exploiting the **shared structure** of biparsers?
- Plan:
  1. *Weaken* round-tripping to be compositional (i.e., property guaranteed by construction).
  2. Find a property that covers the difference between *weak* and “real” round-tripping:

Verifying round-tripping properties

- Baseline, naive verification method: extract parser, extract printer, check that they match.
- Can we do better by exploiting the **shared structure** of biparsers?
- Plan:
  1. *Weaken* round-tripping to be compositional (i.e., property guaranteed by construction).
  2. Find a property that covers the difference between *weak* and “real” round-tripping:
     - necessarily non-compositional,

Verifying round-tripping properties

- Baseline, naive verification method: extract parser, extract printer, check that they match.
- Can we do better by exploiting the **shared structure** of biparsers?
- Plan:
  1. *Weaken* round-tripping to be compositional (i.e., property guaranteed by construction).
  2. Find a property that covers the difference between *weak* and “real” round-tripping:
     - necessarily non-compositional,
     - but hopefully “easier” to verify than real round-tripping.

Backward round-tripping (print-then-parse)

- Recall backward round-tripping:

```
print p a = s   ->   parse p s = Just a
```

Backward round-tripping (print-then-parse)

- Recall backward round-tripping:

```
print p a = s   ->   parse p s = Just a
```

```
parse' :: Biparser x a -> [Char] -> Maybe (a, [Char])
print' :: Biparser x a -> x -> ([Char], a)
```

Backward round-tripping (print-then-parse)

- Recall backward round-tripping:

```
print p a = s   ->   parse p s = Just a
```

```
parse' :: Biparser x a -> [Char] -> Maybe (a, [Char])
print' :: Biparser x a -> x -> ([Char], a)
```

- *Weak backward round-tripping*:

```
print' p x = (s, a)
   ->   parse' p (s ++ s') = Just (a, s')
```

Backward round-tripping (print-then-parse)

- Recall backward round-tripping:

```
print p a = s   ->   parse p s = Just a
```

```
parse' :: Biparser x a -> [Char] -> Maybe (a, [Char])
print' :: Biparser x a -> x -> ([Char], a)
```

- *Weak backward round-tripping*:

```
print' p x = (s, a)
   ->   parse' p (s ++ s') = Just (a, s')
```

  - **Compositional**, i.e., holds by construction.

Compositionality

WBRT: Weak backward round-tripping

- `comap f p` is WBRT, if `p` is WBRT.
- `return a` is WBRT for all `a`
- `(p >>= \a -> k a)` is WBRT, if `p` is WBRT and for all `a`, `k a` is WBRT.

Compositionality

WBRT: Weak backward round-tripping

- `comap f p` is WBRT, if `p` is WBRT.
- `return a` is WBRT for all `a`
- `(p >>= \a -> k a)` is WBRT, if `p` is WBRT and for all `a`, `k a` is WBRT.

Only primitives then need to be checked:

- `biparseChar` is WBRT.

Purification

- Example: printer component of `biparseChar`.

```
type Printer x a = (x -> ([Char], a))
          -- Printer Char Char
printChar :: Char -> ([Char], Char)
printChar c = ([c], c)
--        ^         ^
```

Purification

- Example: printer component of `biparseChar`.

```
type Printer x a = (x -> ([Char], a))
          -- Printer Char Char
printChar :: Char -> ([Char], Char)
printChar c = ([c], c)
--          ^          ^
```

- Key property: **printer returns its input**.

Purification

- Example: printer component of `biparseChar`.

```
type Printer x a = (x -> ([Char], a))
          -- Printer Char Char
printChar :: Char -> ([Char], Char)
printChar c = ([c], c)
--          ^          ^
```

- Key property: **printer returns its input**.

```
-- "Pure projection"
projPrinter :: Printer x a -> (x -> a)
projPrinter q x = let (_, a) = q x in a
```

Purification

- Example: printer component of `biparseChar`.

```
type Printer x a = (x -> ([Char], a))
          -- Printer Char Char
printChar :: Char -> ([Char], Char)
printChar c = ([c], c)
--          ^          ^
```

- Key property: **printer returns its input.**

```
-- "Pure projection"
projPrinter :: Printer x a -> (x -> a)
projPrinter q x = let (_, a) = q x in a
```

- for all `c :: Char`, `projPrinter printChar c = c`
  i.e., `projPrinter printChar = id`

Purification

- Example: printer component of `biparseChar`.

```
type Printer x a = (x -> ([Char], a))
          -- Printer Char Char
printChar :: Char -> ([Char], Char)
printChar c = ([c], c)
--          ^          ^
```

- Key property: **printer returns its input.**

```
-- "Pure projection"
projPrinter :: Printer x a -> (x -> a)
projPrinter q x = let (_, a) = q x in a
```

- for all `c :: Char`, `projPrinter printChar c = c`
  i.e., `projPrinter printChar = id`
- "`printChar` *purifies to* `id`."

Purification

```
-- Let P x a = (x -> a)
--    it's a monad
--    it's a profunctor
--    it's a monadic profunctor
```

Purification

```haskell
-- Let P x a = (x -> a)
--   it's a monad
--   it's a profunctor
--   it's a monadic profunctor
```

- There is a *monadic profunctor morphism*:

```haskell
proj :: Biparser u a -> (u -> a)
```

Purification

```
-- Let P x a = (x -> a)
--   it's a monad
--   it's a profunctor
--   it's a monadic profunctor
```

- There is a *monadic profunctor morphism*:

```
proj :: Biparser u a -> (u -> a)

  proj (return a)         = return a
  proj (p >>= \a -> k a) = proj p >>= \a -> proj (p a)
  proj (comap f p)        = comap f (proj p)
```

Purification

```
-- Let P x a = (x -> a)
--   it's a monad
--   it's a profunctor
--   it's a monadic profunctor
```

- There is a *monadic profunctor morphism*:

```
proj :: Biparser u a -> (u -> a)

  proj (return a)         = return a
  proj (p >>= \a -> k a) = proj p >>= \a -> proj (p a)
  proj (comap f p)        = comap f (proj p)

  proj biparseChar = (id :: Char -> Char)
  proj biparseInt  = (id :: Int -> Int)
```

Purification and backward round-tripping

```
proj :: Biparser u a -> (u -> a)
```

- Biparser `p` *purifies to* `id`: `proj p = id`

Purification and backward round-tripping

```haskell
proj :: Biparser u a -> (u -> a)
```

- Biparser `p` *purifies to* `id`:   `proj p = id`
  - Point: **agnostic** to parser-specific details (i.e., source string manipulations).

Purification and backward round-tripping

```
proj :: Biparser u a -> (u -> a)
```

- Biparser `p` *purifies to* `id`: `proj p = id`
  - Point: **agnostic** to parser-specific details (i.e., source string manipulations).
  - Equational reasoning.

Purification and backward round-tripping

```
proj :: Biparser u a -> (u -> a)
```

- Biparser `p` *purifies to* `id`: `proj p = id`
  - Point: **agnostic** to parser-specific details (i.e., source string manipulations).
  - Equational reasoning.
- Recall *weak backward round-tripping*:

  ```
  print' p x = (s, a)
    ->   parse' p (s ++ s') = Just (a, s')
  ```

  - *Compositional*, i.e., holds by construction.

Purification and backward round-tripping

```
proj :: Biparser u a -> (u -> a)
```

- Biparser `p` *purifies to* `id`: `proj p = id`
  - Point: **agnostic** to parser-specific details (i.e., source string manipulations).
  - Equational reasoning.
- Recall *weak backward round-tripping*:

```
print' p x = (s, a)
   ->   parse' p (s ++ s') = Just (a, s')
```

  - *Compositional*, i.e., holds by construction.
- Weak backward round-tripping $\wedge$ purifies to `id` $\Longrightarrow$ backward round-tripping.

```
print p a = s   ->   parse p s = Just a
```

Forward round-tripping (parse-then-print)

- *Weak forward round-tripping*

```
parse' p s = Just (a, s'') -- and
print' p x = (a, s')       --
    ->    s = s' ++ s''
```

Forward round-tripping (parse-then-print)

- *Weak forward round-tripping*

```
parse' p s = Just (a, s'') -- and
print' p x = (a, s')       --
   ->   s = s' ++ s''
```

  - **Quasicompositional**: some side conditions to satisfy!?

Forward round-tripping (parse-then-print)

- *Weak forward round-tripping*

```
parse' p s = Just (a, s'') -- and
print' p x = (a, s')       --
    ->   s = s' ++ s''
```

  - **Quasicompositional**: some side conditions to satisfy!?
- Weak forward round-tripping $\wedge$ purifies to `id` $\Longrightarrow$ forward round-tripping.

Compositionality (recall)

WBRT: Weak backward round-tripping

- `comap f p` is WBRT, if `p` is WBRT.
- `return a` is WBRT for all `a`
- `(p >>= \a -> k a)` is WBRT, if `p` is WBRT and for all `a`, `k a` is WBRT.

Quasicompositionality

WFRT: Weak forward round-tripping

- `comap f p` is WFRT, if `p` is WFRT.
- `return a` is WFRT for all `a`
- `(p >>= \a -> k a)` is WFRT, if `p` is WFRT and for all `a`, `k a` is WFRT, **and `k` is an injective arrow**.

Quasicompositionality

WFRT: Weak forward round-tripping

- `comap f p` is WFRT, if `p` is WFRT.
- `return a` is WFRT for all `a`
- `(p >>= \a -> k a)` is WFRT, if `p` is WFRT and for all `a`, `k a` is WFRT, **and `k` is an injective arrow**.

- Injectivity generalized to Kleisli arrows.

Quasicompositionality

WFRT: Weak forward round-tripping

- `comap f p` is WFRT, if `p` is WFRT.
- `return a` is WFRT for all `a`
- `(p >>= \a -> k a)` is WFRT, if `p` is WFRT and for all `a`, `k a` is WFRT, **and `k` is an injective arrow**.

- Injectivity generalized to Kleisli arrows.
- `k :: v -> m w` is an *injective arrow* if there exists a function `k' :: w -> v` such that:

```
  k x >>= (\y -> return (   x, y))
= k x >>= (\y -> return (k' y, y))
```

Quasicompositionality: example

- The function

  ```
  (\ n -> replicateP n p)
    :: Int -> Biparser [Char] [Char]
  ```

  is an injective arrow, and `length :: [Char] -> Int` is its *sagittal inverse*.

```
  replicateP n p >>= (\xs -> return (         n, xs))
= replicateP n p >>= (\xs -> return (length xs, xs))
```

Summary

Summary

- Monads for bidirectional programming: *monadic profunctors*.

Summary

- Monads for bidirectional programming: *monadic profunctors*.
- Round-tripping decomposed into *weak round-tripping* and a *purification* property.
  - Only need to reason about a domain-agnostic interpretation of the program.

Summary

- Monads for bidirectional programming: *monadic profunctors*.
- Round-tripping decomposed into *weak round-tripping* and a *purification* property.
  - Only need to reason about a domain-agnostic interpretation of the program.
- Problem in the parse-then-print round-trip: *generalized injectivity* requirement.

Summary

- Monads for bidirectional programming: *monadic profunctors*.
- Round-tripping decomposed into *weak round-tripping* and a *purification* property.
  - Only need to reason about a domain-agnostic interpretation of the program.
- Problem in the parse-then-print round-trip: *generalized injectivity* requirement.
- More in the paper: lenses and random generators-predicates.

Conclusion

Future work:

- More practice, more features, e.g., backtracking, lookahead in parsers?[1]

[1] https://github.com/Lysxia/unparse-attoparsec

Conclusion

Future work:

- More practice, more features, e.g., backtracking, lookahead in parsers?[1]
- How to enforce injectivity of arrows/functions (maybe linear types)?

[1] `https://github.com/Lysxia/unparse-attoparsec`

Conclusion

Future work:

- More practice, more features, e.g., backtracking, lookahead in parsers?[1]
- How to enforce injectivity of arrows/functions (maybe linear types)?
- A theory of bidirectional programs with round-tripping properties? (`Fwd m`, `Bwd n`)

[1] `https://github.com/Lysxia/unparse-attoparsec`

Conclusion

Future work:

- More practice, more features, e.g., backtracking, lookahead in parsers?[1]
- How to enforce injectivity of arrows/functions (maybe linear types)?
- A theory of bidirectional programs with round-tripping properties? (`Fwd m`, `Bwd n`)

Thank you!

[1] `https://github.com/Lysxia/unparse-attoparsec`