Biparsers: Exact-Printing for Data Synchronisation

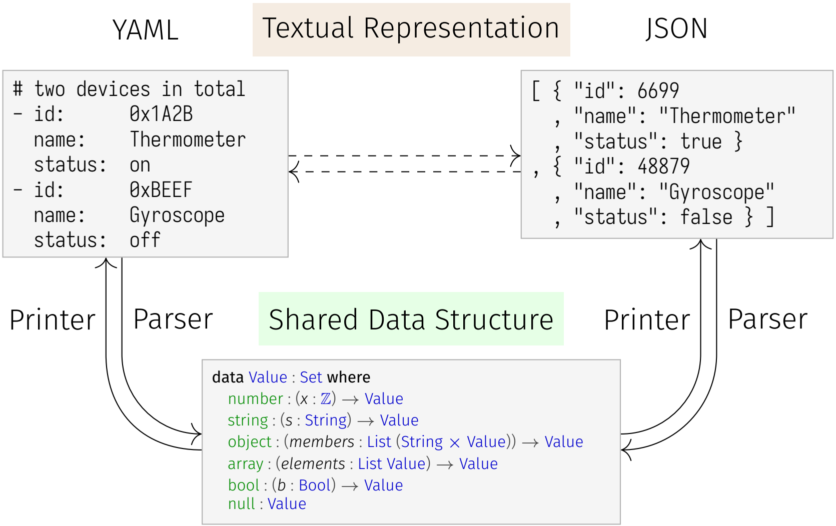
Ruifeng Xie[1,2] Tom Schrijvers[2] Zhenjiang Hu[1]

1. Peking University (School of Computer Science)
2. KU Leuven (Department of Computer Science)

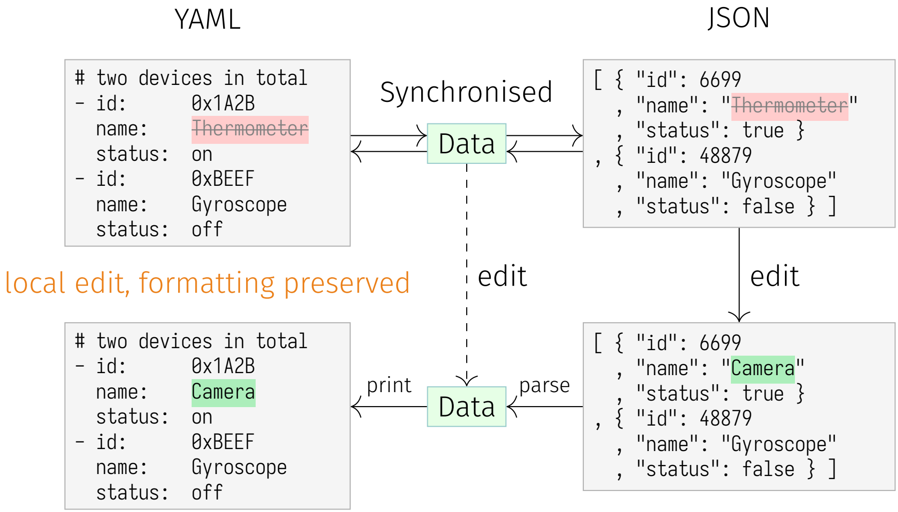
January 23, 2025 @ POPL '25

Problem: Data Synchronisation

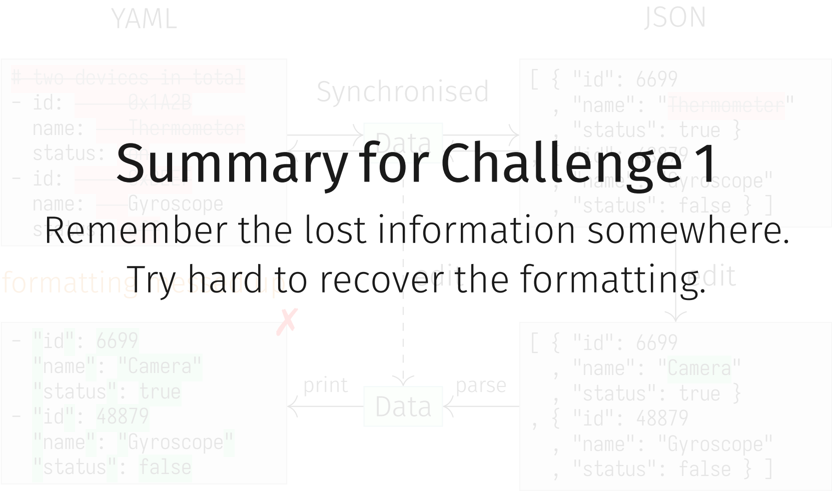
Example: Configuration Synchronisation



Synchronisation: Expected Behaviour



Challenge 1: Non-Injectivity (Exact-Printing)



Summary for Challenge 1

Remember the lost information somewhere.

Try hard to recover the formatting.

Challenge 2: Parser-Printer Consistency

Summary for Challenge 2

Do not try too hard to recover the formatting.

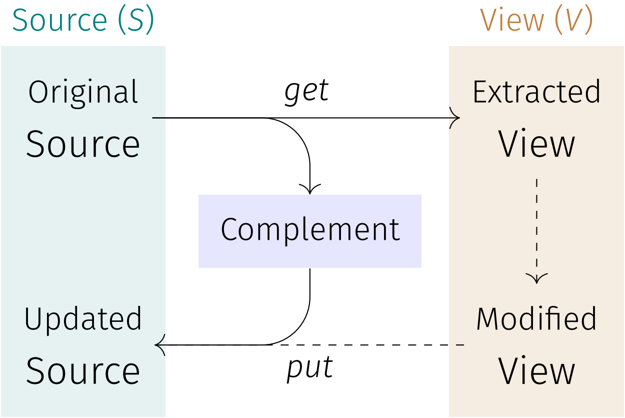
Be careful when the text would change meaning.

Solution: Biparsers

Bidirectional Transformation (Lens)

[Foster 2007; Hofmann et al. 2011]

Parsing/printing aside, we already know how to synchronise data.



Non-Injectivity

get is not injective.

Cannot put with view alone.

Complement

Save the lost information.

put with complement.

Biparsers: Generalisation of Lenses

Lenses are pure; they cannot handle parsing/printing.

For composable parsing/printing, we need the P monad (state + error).

```
record P (A : Set) : Set where
  field runP : String → Result Error (A × String)
```

Biparsers are lenses generalised with monadic parsing and printing:

```
record Biparser (C X Y : Set) : Set where
  field parse : X → P (Y × C)
        print : Y × C → P X
```

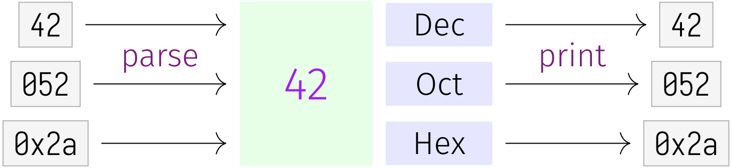
```
get : X → Y × C
put : Y × C → X
```

Existing definitions and properties for lenses also need to be generalised.

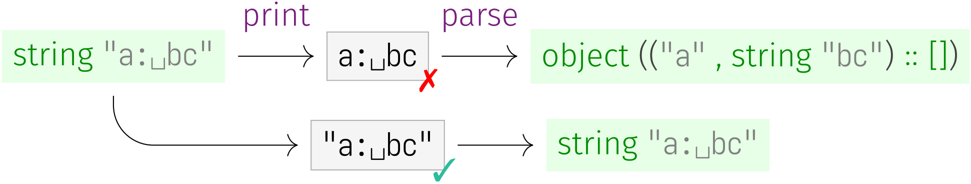
Generalised Round-Trip Properties

Generalising round-trip properties ⇔ addressing the two challenges.

- Challenge 1 (Exact-Printing): parsed data can be printed back

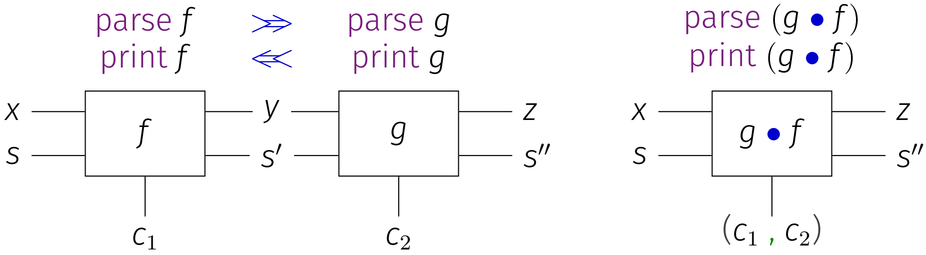


- Challenge 2 (Consistency): printed text can be parsed back



Generalised Composition

Composition uses Kleisli composition and accumulates the complement:



- Composition = Sequencing (p then q) + Transformation ($x \Rightarrow y \Rightarrow z$).
- Main result: composition preserves round-trip properties.

Compositional: combine small biparsers into larger biparsers.

Round-trip properties are automatically derived.

A Rich Combinator Language

Biparser Combinators

Classical parser combinator framework (e.g., Parsec) [Wadler 1995]:

- *item*: token consumption
- Sequencing: first p then q
- Alternation (Choice): either p or q
- Filtering: validate and transform

Special Care for Round-Trip Properties

Composition = Sequencing + Transform

By analogy, we define biparser combinators.

Main Challenges

- Take complements into consideration
- Make sure combinators preserve round-trip properties

Example: JSON Strings (Parser Combinators)

Parser resembles the grammar

Sequencing
Alternation
Transformation

$\langle String \rangle ::=$ `"` $\langle StringChar \rangle^*$ `"`

```
string = char '"' *> many stringChar <* char '"'
```

$\langle StringChar \rangle ::=$ NONCONTROL | `\` $\langle CEsc \rangle$ | `\` $\langle UniEsc \rangle$

```
stringChar = nonControl <|> (char '\\' *> cEsc) <|> (char '\\' *> uniEsc)
```

$\langle CEsc \rangle ::=$ `"` | `\` | `/` | `b` | `f` | `n` | `r` | `t`

```
cEsc =    (const '"'  <$ char '"') <|> (const '\\' <$ char '\\')
      <|> (const '/'  <$ char '/') <|> (const '\b' <$ char 'b')
      <|> (const '\f' <$ char 'f') <|> (const '\n' <$ char 'n')
      <|> (const '\r' <$ char 'r') <|> (const '\t' <$ char 't')
```

$\langle UniEsc \rangle ::=$ `u` HEX HEX HEX HEX

```
uniEsc = char 'u' *> (mkUnicodeChar <$> hex <*> hex <*> hex <*> hex)
```

Example: JSON Strings (Biparsers)

Biparser resembles the grammar
Biparser resembles the parser
Simple Development and Migration

⟨*String*⟩ ::= " ⟨*StringChar*⟩* "

```
string = char '"' #*≫ many stringChar ≪*# char '"'
```

⟨*StringChar*⟩ ::= NonControl | \ ⟨*CEsc*⟩ | \ ⟨*UniEsc*⟩

```
stringChar = nonControl <+|> (char '\\' #*≫ cEsc) <+|> (char '\\' #*≫ uniEsc)
```

⟨*CEsc*⟩ ::= " | \ | / | b | f | n | r | t

```
cEsc =     (const' '"'  #• char '"') <+> (const' '\\' #• char '\\')
       <+> (const' '/'  #• char '/') <+> (const' '\b' #• char 'b')
       <+> (const' '\f' #• char 'f') <+> (const' '\n' #• char 'n')
       <+> (const' '\r' #• char 'r') <+> (const' '\t' #• char 't')
```

⟨*UniEsc*⟩ ::= u Hex Hex Hex Hex

```
uniEsc = char 'u' #*≫ (mkUnicodeChar #• (hex ≪*≫ hex ≪*≫ hex ≪*≫ hex))
```

Highlights of Our Solution

Compatibility between Lenses and Effects

Previous Work [Abou-Saleh et al., 2015]

Serious restrictions on how effects can be used in bidirectional programs.

We solve the problem for parser/printer effects:

Complement-Based Lenses

- Avoids the get in lens compositions.
- Makes composition preserve the round-trip properties.
- Exposes complements for reasoning; improves printing performance.

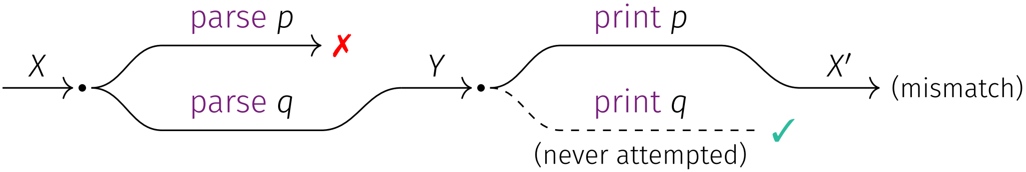
Effect Cancellation

- print *rewinds* the effects performed by parse (and vice versa).
- Fits our purpose perfectly and simplifies the theory.

Backtracking Choice and Consistency Hazard

Parser combinators rely on (shallow) backtracking for choices.

"*p* or *q*": attempt *p*, and if *p* fails, attempt *q*.



Solution: different choice combinators with different pre-conditions.

More Notable Features

- Printing from scratch: optional complement.
- Context sensitivity: support for arbitrary user state.
- Improving output quality: view-complement alignment.
- Fine-grained complement manipulation.

Proof and Example

Agda Proof and Haskell Example

Mechanised Proof in Agda

- Around 1300 LoC (including comments and blank lines).
- All combinators with round-trip properties verified.

Runnable Examples in Haskell (JSON and YAML Subsets)

- Around 1000 LoC for combinators, 300 LoC for each example.
- All combinators with full Haddock documentation.

Both available as reusable packages.

Conclusion

Conclusion

- Biparsers with *exact-printing*: lens + monadic parsing/printing.
  - Formalised consistency: round-trip properties.
  - Compositional programming:
    Composition preserves round-trip properties.
- Rich combinator language: biparser combinators.

Artifact available as reusable packages:

- Mechanised proof in Agda for the combinators.
- Runnable examples in Haskell for JSON and YAML.

More details in the paper!

Outline for Navigation

Outline

Problem: Data Synchronisation

Solution: Biparsers

A Rich Combinator Language

Highlights of Our Solution

Proof and Example

Conclusion

Details

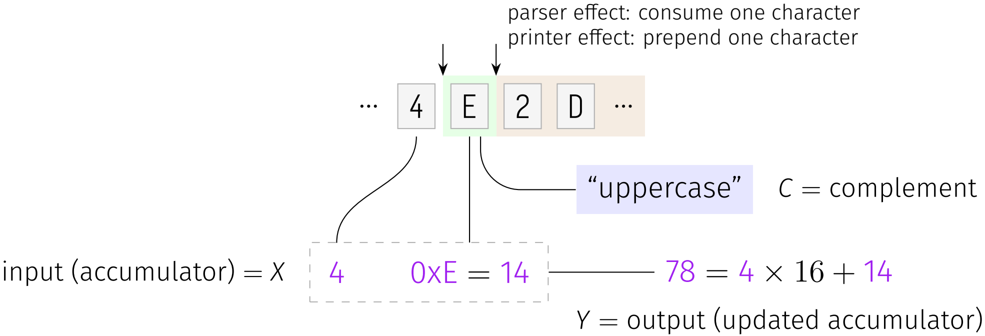
FAQ

1. What are X, Y, and C in practical biparsers?
2. Round-Trip Properties: What happens when the view is modified?
   View-Complement Alignment: How does it work?
3. Can you elaborate on your choice combinators?
4. How do you design the complements?
   Do you define a data type for the complement of every biparser?

Biparsers: Intuition

Two aspects of biparsers: parsing/printing + transformation.

Consider a biparser for hexadecimal numbers.



Round-Trip Properties, for Modified Text/Data

Edits do not void the guarantee of round-trip properties.

Power of Compositional Programming

- Biparsers are constructed compositionally.
- Component biparsers synchronise between text segments and subtrees.
- Round-trip properties apply to unchanged segments and subtrees.

Can we do even better?

- Before printing, perform a view-complement alignment.
- Generally applicable to any biparser.
- Various reusable strategies, orthogonal to parsing/printing.
- Only affects output quality, correctness always guaranteed.

Back to FAQ

Choice and the Consistency Conditions

Two Types of Backtracking Choices

- Biased choice: left branch preferred (the *regular* version).
- Remembered choice: the branch used last time is preferred.

Consistency Conditions

- Non-Intersection: disjoint branches (= unambiguous grammar).
- Consistency: same behaviour for intersection.
- Weak Consistency: consistency, under certain circumstances.

Back to FAQ

Complement Types

Complement types are not designed, but discovered.

- Follow the grammar and write the biparser.
- The suitable complement emerges as a result.

Fine-grained control over complements is also possible:

- Combinators dedicated to complement manipulation: mapC etc.
- Define complement data types for readability.

◂ Back to FAQ