# PisoLang: a User-Friendly Reversible Programming Language with Inductive Types

July 10, 2026

Kosuke Onodera, Keisuke Nakano, Kazuyuki Asada, and Kentaro Kikuchi

Tohoku University, Japan

# Reversible Programming

Programming where computation can be reversed

```
a += 3          b += a
b -= a    ⟷     a -= 3
```

## Application of Reversible Computation

- Low-power electronics

  Any logically irreversible process is accompanied by entropy production [Landauer '61]

## Application of Reversible Computation

- Low-power electronics

  Any logically irreversible process is accompanied by entropy production [Landauer '61]

- Encoding/decoding

  $\texttt{encode}\quad \texttt{"Hello"} \quad \Rightarrow^* \quad \texttt{"Ifmmp"}$

  $\texttt{encode}^{-1}\ \texttt{"Ifmmp"} \quad \Rightarrow^* \quad \texttt{"Hello"}$

## Application of Reversible Computation

- Low-power electronics

  Any logically irreversible process is accompanied by entropy production [Landauer '61]

- Encoding/decoding

  $\texttt{encode}\quad \texttt{"Hello"} \quad \Rightarrow^* \quad \texttt{"Ifmmp"}$

  $\texttt{encode}^{-1}\ \texttt{"Ifmmp"} \quad \Rightarrow^* \quad \texttt{"Hello"}$

- Database recovery
- Testing/verification

  *Time-travel debugging* via backward execution

## Prior Work [Chardonnet+ '24]

A reversible functional programming language with inductive types focused on:

- Expressivity: *reversible Turing complete*
- Denotational semantics (and adequate operational semantics)

## Prior Work [Chardonnet+ '24]

A reversible functional programming language with inductive types focused on:

- Expressivity: *reversible Turing complete*
- Denotational semantics (and adequate operational semantics)

PisoLang is heavily inspired by the language

*Map* function on lists in [Chardonnet+ '24]

```
fix map. \f. {
  | fold (injl ()) ⟷ fold (injl ())
  | fold (injr (x, xs)) ⟷
      let x'  = f x in
      let xs' = map f xs in
      fold (injr (x', xs'))
}
```

## This Work (User-Friendliness) (1/2)

We extend that language with

# This Work (User-Friendliness) (1/2)

We extend that language with

- A more relaxed surface syntax

```
iso rec map f = case
| []      ⟷ []
| x :: xs ⟷ f x :: map f xs
```

Originally:

```
fix map. \f. {
  | fold (injl ()) ⟷ fold (injl ())
  | fold (injr (x, xs)) ⟷
      let x'  = f x in
      let xs' = map f xs in
      fold (injr (x', xs'))
}
```

# This Work (User-Friendliness) (1/2)

We extend that language with

- A more relaxed surface syntax

```
iso rec map f = case
| []      ⟷ []
| x :: xs ⟷ f x :: map f xs
```

Originally:

```
fix map. \f. {
  | fold (injl ()) ⟷ fold (injl ())
  | fold (injr (x, xs)) ⟷
      let x'  = f x in
      let xs' = map f xs in
      fold (injr (x', xs'))
}
```

- Hindley-Milner-style type inference and polymorphism

```
map : ('a ⟷ 'b) → ('a list ⟷ 'b list)
```

# This Work (User-Friendliness) (1/2)

We extend that language with

- A more relaxed surface syntax

```
iso rec map f = case
| []      ⟷ []
| x :: xs ⟷ f x :: map f xs
```

Originally:

```
fix map. \f. {
  | fold (injl ()) ⟷ fold (injl ())
  | fold (injr (x, xs)) ⟷
      let x'  = f x in
      let xs' = map f xs in
      fold (injr (x', xs'))
}
```

- Hindley-Milner-style type inference and polymorphism

```
map : ('a ⟷ 'b) → ('a list ⟷ 'b list)
```

Operational semantics is also naturally extended, and has

- Subject reduction
- Well-behavedness of inversions ($\omega\ v \Rightarrow^* v'$ iff $\omega^{-1}\ v' \Rightarrow^* v$)

## This Work (User-Friendliness) (2/2)

- User-defined types
  - E.g., `nat` for natural numbers; `'a list` for lists of elements of type `'a`

```
type      nat = Z | S of nat
type 'a list = Nil | Cons of 'a
```

## This Work (User-Friendliness) (2/2)

- User-defined types
  - E.g., `nat` for natural numbers; `'a list` for lists of elements of type `'a`

```
type      nat = Z | S of nat
type 'a list = Nil | Cons of 'a
```

  - E.g., a list of natural numbers

PisoLang

```
Cons (Z, Cons (S Z, Nil))
```

or even `[0; 1]`

Originally...

```
fold injr (fold injl (), fold injr (
fold injr fold injl (), fold injl ()))
```

## This Work (User-Friendliness) (2/2)

- User-defined types
  - E.g., `nat` for natural numbers; `'a list` for lists of elements of type `'a`

```
type      nat = Z | S of nat
type 'a list = Nil | Cons of 'a
```

  - E.g., a list of natural numbers

PisoLang

```
Cons (Z, Cons (S Z, Nil))
```

or even `[0; 1]`

Originally...

```
fold injr (fold injl (), fold injr (
fold injr fold injl (), fold injl ()))
```

- Pattern-based duplication/erasure

```
iso erase (x, x) = x
```

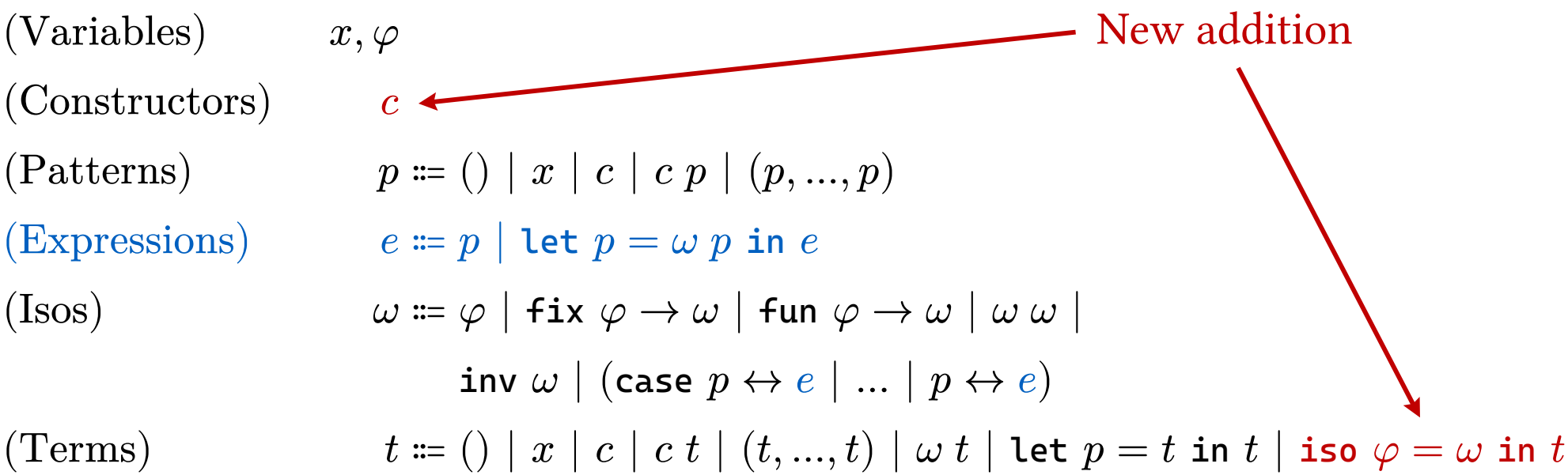
E.g., `erase (3, 3)` $\Rightarrow^*$ `3`

(Originally done by recursive macros)

## The Rest of This Talk

- Syntax and types
  - Grammar
  - Reversible pattern matching
  - Base types and *iso* types
- More details on typing rules, time permitting
- Related Work
- Conclusion

## Grammar (Abridged)




- Expressions: right-hand sides of branches
- *Isos*: functions, named after (partial) *iso*morphisms

# Reversible Pattern Matching - *add*

```
type nat = Z | S of nat

(* add (x, y) = (x + y, y) *)
iso rec add = case
| (m, 0)   ⟷ (m, 0)
| (m, S n) ⟷ let (m', n') = add (S m, n) in (m', S n')
in

add (3, 4) (* = (7, 4) *)
```

A program consists of type definitions followed by a single term

# Reversible Pattern Matching - *add*

```
type nat = Z | S of nat

(* add (x, y) = (x + y, y) *)
iso rec add = case
| (m, 0)   ⟷ (m, 0)
| (m, S n) ⟷ let (m', n') = add (S m, n) in (m', S n')
in

add (3, 4) (* = (7, 4) *)
```

A program consists of type definitions followed by a single term

The built-in iso `inv` inverts any well-typed iso

```
inv add (7, 4) (* = (3, 4) *)
```

## Reversible Pattern Matching - Syntax inherited from [Chardonnet+ '24]

$$
\begin{array}{l}
\left(\begin{array}{l}
\texttt{case} \\
\quad \mid p_1 \leftrightarrow \texttt{let } p_{11} = \omega_{11}\ p'_{11} \texttt{ in} \\
\qquad\quad \cdots \\
\qquad\quad \texttt{let } p_{1n} = \omega_{1n}\ p'_{1n} \texttt{ in} \\
\qquad\quad p'_1 \\
\quad \mid p_2 \leftrightarrow \ldots \\
\cdots \\
\quad \mid p_n \leftrightarrow \ldots
\end{array}\right)^{-1} \\
:= \\
\begin{array}{l}
\texttt{case} \\
\quad \mid p'_1 \leftrightarrow \texttt{let } p'_{1n} = \omega_{1n}^{-1}\ p_{1n} \texttt{ in} \\
\qquad\quad \cdots \\
\qquad\quad \texttt{let } p'_{11} = \omega_{11}^{-1}\ p_{11} \texttt{ in} \\
\qquad\quad p_1 \\
\quad \mid p'_2 \leftrightarrow \ldots \\
\cdots \\
\quad \mid p'_n \leftrightarrow \ldots
\end{array}
\end{array}
$$

$p$: pattern $\omega$: iso

Inversion is defined structurally

# Reversible Pattern Matching - Restrictions inherited from [Chardonnet+ '24]

```
iso rec add = case
| (m, 0)   ⟷                                  (m, 0)
| (m, S n) ⟷ let (m', n') = add (S m, n) in (m', S n')
```

# Reversible Pattern Matching - Restrictions inherited from [Chardonnet+ '24]

```
iso rec add = case
| (m, 0)   ⟷                                  (m, 0)
| (m, S n) ⟷ let (m', n') = add (S m, n) in (m', S n')
```

- Non-overlap

Non-overlap here as well

Ensures injectivity; patterns may be non-exhaustive ⇒ partiality

# Reversible Pattern Matching - Restrictions inherited from [Chardonnet+ '24]

```
iso rec add = case
| (m, 0)   ⟷                                  (m, 0)
| (m, S n) ⟷ let (m', n') = add (S m, n) in (m', S n')
```

- Non-overlap — Non-overlap here as well

  Ensures injectivity; patterns may be non-exhaustive ⇒ partiality

- Linear variable use and no free variables

  Error (non-linear use of x) `x ⟷ let y = x in (x, y)`

  Error (free occurrence of z) `x ⟷ let y = z in (x, y)`

  Inverse isos are well-typed and well-behaved (i.e., $(\texttt{inv}\ \omega) \circ \omega \sqsubseteq \mathrm{id}$)

## Example - *fib*

- `fib` $n$ computes the $(n+1)$-th and $(n+2)$-th Fibonacci numbers

```
(* add (m, n) = (m + n, n) *)
iso rec fib = case
| 0   ⟷ (1, 1)
| S n ⟷ let (S (S m'), n') = add (fib n) in (n', S (S m'))
```

Non-overlap

## Example - *fib*

- `fib` $n$ computes the $(n+1)$-th and $(n+2)$-th Fibonacci numbers

```
(* add (m, n) = (m + n, n) *)
iso rec fib = case
| 0   ⟷ (1, 1)
| S n ⟷ let (S (S m'), n') = add (fib n) in (n', S (S m'))
```

Non-overlap

- Error

```
iso rec fib = case
| 0   ⟷ (1, 1)
| S n ⟷ let (m', n') = add (fib n) in (n', m')
```

Overlap

## Repository and More Examples

See https://github.com/42067/reversible_lang for more examples:

- `README.md` (introduces many features including syntactic sugar)
- Reversible Turing machines
- Polymorphic reversible merge sort
- Polymorphic run-length compression
- Prime factorization

## Reversible Pattern Matching - Automatic Expansion

- One may write:

```
iso compose f g = case
| x ⟷ f (g x)
```

## Reversible Pattern Matching - Automatic Expansion

- One may write:

```
iso compose f g = case
| x ⟷ f (g x)
```

- The code is expanded into the core language as follows:

```
iso compose f g = case
| x ⟷ let _0 = g x  in
       let _1 = f _0 in
       _1
```

# Reversible Pattern Matching - Pattern-Based Duplication

- Language-level duplication of values via patterns (new addition)

```
iso rec double = case
| []      ⟷ []
| x :: xs ⟷ x :: x :: double xs
```

- Duplication

  `double [A; B; C]` $\Rightarrow^*$ `[A; A; B; B; C; C]`

# Reversible Pattern Matching - Pattern-Based Duplication

- Language-level duplication of values via patterns (new addition)

```
iso rec double = case
| []      ⟷ []
| x :: xs ⟷ x :: x :: double xs
```

- Duplication

  `double [A; B; C]` $\Rightarrow^*$ `[A; A; B; B; C; C]`

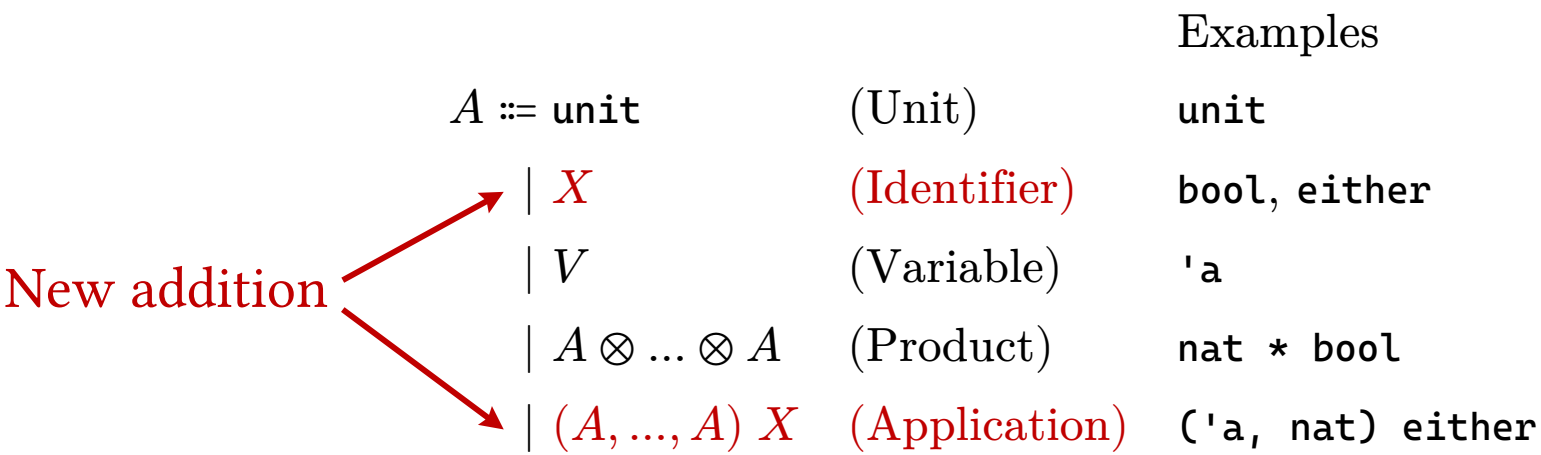
- Originally written as follows:

```
fix double. {
  | fold (injl ()) ⟷ fold (injl ())
  | fold (injr (x, xs)) ⟷
      let (x1, x2) = Dup x in
      let xs' = double xs in
      fold (injr (x1, fold injr (x2, xs'))) }
```

A macro that duplicates any value
(required in the proof of r-Turing completeness)

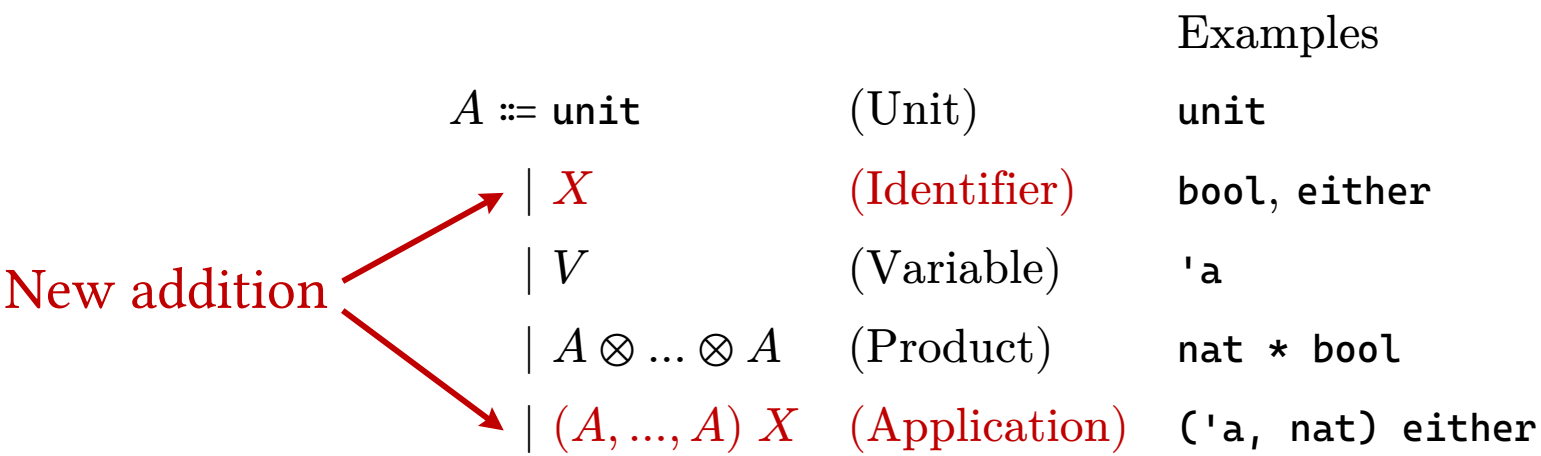
# Types

- Base types: non-functional types



# Types

- Base types: non-functional types




- *Iso* types: function types

$T ::= A \leftrightarrow A$ (Reversible; for `case`) `bool ⟷ nat`

$\mid T \to T$ (Irreversible; for higher-order fn.) `(bool ⟷ nat) → (unit ⟷ nat)`

## The Rest of This Talk

- Syntax and types
  - Grammar
  - Reversible pattern matching
  - Base types and *iso* types
- More details on typing rules, time permitting
- Related Work
- Conclusion

## Typing Judgments - Overview

- Isos (non-linear) $\Gamma \vdash_{\omega} \omega : T$
- Terms (non-linear) $\Gamma; \Delta \vdash t : A$
- Expressions (linear; responsible for reversibility) $\Gamma; \Delta \vdash_{\mathrm{e}} e : A$

where $\Gamma$: ctx. for iso-typed terms and $\Delta$: ctx. for base-typed terms

# Typing Judgments - Expressions (Abridged)

- A pattern may have more than one occurrence of the same variable

$$\frac{}{\Gamma; \varnothing \vdash_{\mathrm{e}} () : \mathtt{unit}}\ (\text{X-Unit}) \qquad \frac{}{\Gamma; x : A \vdash_{\mathrm{e}} x : A}\ (\text{X-Var})$$

$$\frac{\varnothing; \Delta \vdash_{\mathrm{e}} p_1 : A_1 \quad \ldots \quad \varnothing; \Delta \vdash_{\mathrm{e}} p_n : A_n}{\Gamma; \Delta \vdash_{\mathrm{e}} (p_1, \ldots, p_n) : A_1 \otimes \ldots \otimes A_n}\ (\text{X-Tuple})$$

# Typing Judgments - Expressions (Abridged)

- A pattern may have more than one occurrence of the same variable

$$\frac{}{\Gamma; \varnothing \vdash_{\mathrm{e}} () : \mathtt{unit}}\ (\text{X-Unit}) \qquad \frac{}{\Gamma; x : A \vdash_{\mathrm{e}} x : A}\ (\text{X-Var})$$

$$\frac{\varnothing; \Delta \vdash_{\mathrm{e}} p_1 : A_1 \quad \ldots \quad \varnothing; \Delta \vdash_{\mathrm{e}} p_n : A_n}{\Gamma; \Delta \vdash_{\mathrm{e}} (p_1, \ldots, p_n) : A_1 \otimes \ldots \otimes A_n}\ (\text{X-Tuple})$$

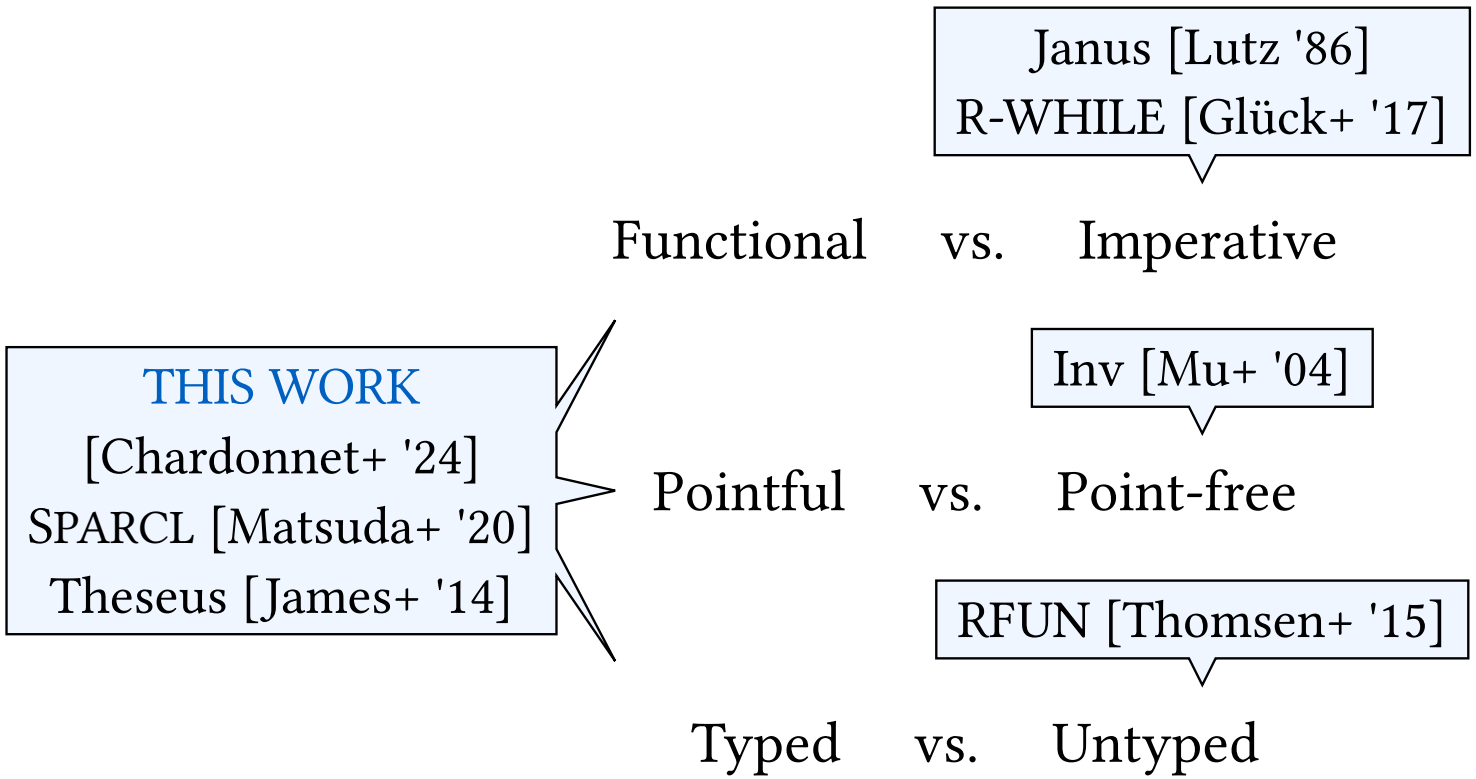
- Variables appearing in $p_2$ cannot be used in $e$

$$\frac{\begin{array}{c}\{x_1, \ldots, x_n\} = \mathrm{FV}(p_1) \quad \varnothing; x_1 : A_1, \ldots, x_n : A_n \vdash_{\mathrm{e}} p_1 : B \quad \Gamma \vdash_{\omega} \omega : A \leftrightarrow B \\ \varnothing; \Delta' \vdash_{\mathrm{e}} p_2 : A \quad \Gamma; \Delta, x_1 : A_1, \ldots, x_n : A_n \vdash_{\mathrm{e}} e : C\end{array}}{\Gamma; \Delta, \Delta' \vdash_{\mathrm{e}} \mathtt{let}\ p_1 = \omega\ p_2\ \mathtt{in}\ e : C}\ (\text{X-Let})$$

## The Rest of This Talk

- Syntax and types
  - Grammar
  - Reversible pattern matching
  - Base types and *iso* types
- More details on typing rules, time permitting
- Related Work
- Conclusion

# Related Work - Design Choices



## Related Work - RFUN v2 [Thomsen+ '15]

- The most similar language to PisoLang in terms of design

## Related Work - RFUN v2 [Thomsen+ '15]

- The most similar language to PisoLang in terms of design
- Differences we observe from their implementation:

(No formal specification was found as far as we know)

| | Types | Injectivity checker | Erasure of equal values via |
|---|---|---|---|
| **RFUN v2** | Explicit | Dynamic | Built-in function |
| **PisoLang** | Inferred | Static | Pattern |

## The Rest of This Talk

- Syntax and types
  - Grammar
  - Reversible pattern matching
  - Base types and *iso* types
- More details on typing rules, time permitting
- Related Work
- Conclusion

## Conclusion

We extended the reversible language of Chardonnet et al. so that:

## Conclusion

We extended the reversible language of Chardonnet et al. so that:

- Reversible functions can be written with less effort
  - Type inference with parametric polymorphism
  - User-defined type/data constructors
  - A relaxed grammar supported by automatic expansion

## Conclusion

We extended the reversible language of Chardonnet et al. so that:

- Reversible functions can be written with less effort
  - Type inference with parametric polymorphism
  - User-defined type/data constructors
  - A relaxed grammar supported by automatic expansion
- The following basic properties are preserved:
  - Subject reduction
  - Well-behavedness of inversions

## Conclusion

We extended the reversible language of Chardonnet et al. so that:

- Reversible functions can be written with less effort
  - Type inference with parametric polymorphism
  - User-defined type/data constructors
  - A relaxed grammar supported by automatic expansion
- The following basic properties are preserved:
  - Subject reduction
  - Well-behavedness of inversions
- New properties are established
  - Soundness/completeness of type inference

Proven after submission :)

## Future Work

- Applications to involutions

  An involution, a function that is its own inverse, can be constructed from injective functions

- Denotational semantics

  As in [Chardonnet+ '24]

- A type system inspired by polymorphic variants

```
iso incr = case
| n ⟷ S n
```

  Should `incr` have type `nat ⟷ [< S of nat]`?

## Extra (Eta-expansion)

```
fun f → inv f
  : 'A → ~'A
```

- Non-keyword inversion by eta-expanding

# Extra (Inversion of Isos)

**Definition (Inversion of isos).** *$\omega^{-1}$, the inverse of an iso $\omega$, is defined as:*

$$
\begin{array}{rcl}
\varphi^{-1} & := & \varphi \\
(\texttt{fix}\ \varphi \to \omega)^{-1} & := & \texttt{fix}\ \varphi \to \omega^{-1} \\
(\texttt{fun}\ \varphi \to \omega)^{-1} & := & \texttt{fun}\ \varphi \to \omega^{-1} \\
(\omega_1\ \omega_2)^{-1} & := & \omega_1^{-1}\ \omega_2^{-1} \\
(\texttt{inv}\ \omega)^{-1} & := & \texttt{inv}\ \omega^{-1} \\
(\texttt{case}\ p_1 \leftrightarrow e_1 \mid \ldots \mid p_n \leftrightarrow e_n)^{-1} & := & \texttt{case}\ (p_1 \leftrightarrow e_1)^{-1} \mid \ldots \mid (p_n \leftrightarrow e_n)^{-1}
\end{array}
$$

$$
\text{where} \begin{pmatrix} p \leftrightarrow \texttt{let}\ p_1 = \omega_1\ p_1'\ \texttt{in} \\ \quad \ldots \\ \quad \texttt{let}\ p_n = \omega_n\ p_n'\ \texttt{in} \\ \quad p' \end{pmatrix}^{-1} := \begin{array}{l} p' \leftrightarrow \texttt{let}\ p_n' = \omega_n^{-1}\ p_n\ \texttt{in} \\ \quad \ldots \\ \quad \texttt{let}\ p_1' = \omega_1^{-1}\ p_1\ \texttt{in} \\ \quad p \end{array}
$$

# Extra (Inversion of Types)

**Definition (Inversion of types).** *$T^{-1}$, the inverse type of an iso type $T$, is defined as:*

$$\begin{aligned}(A \leftrightarrow B)^{-1} &:= B \leftrightarrow A \\ (T_1 \to T_2)^{-1} &:= T_1^{-1} \to T_2^{-1}\end{aligned}$$