SPARCL

A Language for Partially Invertible Computation

Kazutaka Matsuda
Tohoku University, Japan

Meng Wang
University of Bristol, UK

PIP Here

a **s**ystem for **pa**rtially-**r**eversible **c**omputation with **l**inear types

Sparcl

A Language for Partially Invertible Computation

Kazutaka Matsuda

Tohoku University, Japan

Meng Wang

University of Bristol, UK

PIP Here

Background: Invertible Programming

- Invertibility is common in software development
  - compression/decompression
  - undo/redo
  - serialization/deserialization
- Invertible programming provides correctness by construction
  - ***What should be the building blocks?***

PIP Here

Building Blocks?

- Candidate 1: invertible functions
  - injective, thus restrictive
- Candidate 2: ***partially-invertible*** functions
  - functions become invertible by fixing some arguments
    - addition (e.g., $\lambda n.\ n + 42$ is invertible)
    - Huffman encoding

Language for partially-invertible programming?

PIP Here

Our Answer: SPARCL

- A programming language ...
  - for writing ***invertible*** functions
  - through composing ***partially-invertible*** functions

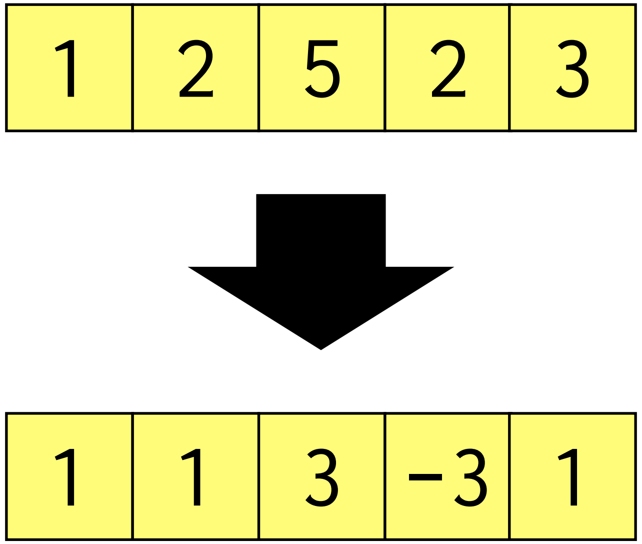
more natural and more expressive

  - supported by ***linear types***
    - a key to ***correctness by construction***

PIP Here

Running Example

- Differences of adjacent elements in a list

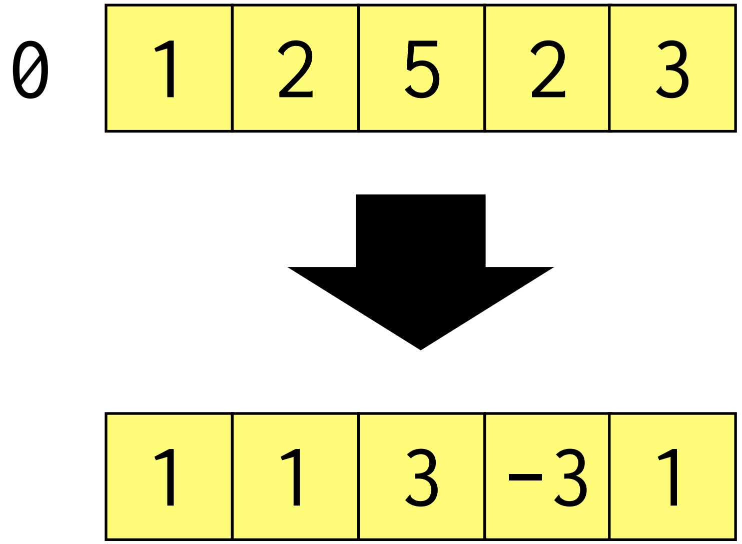


  - a pre-processing for image compression
    - cf. PNG

PIP Here

Running Example

- Differences of adjacent elements in a list



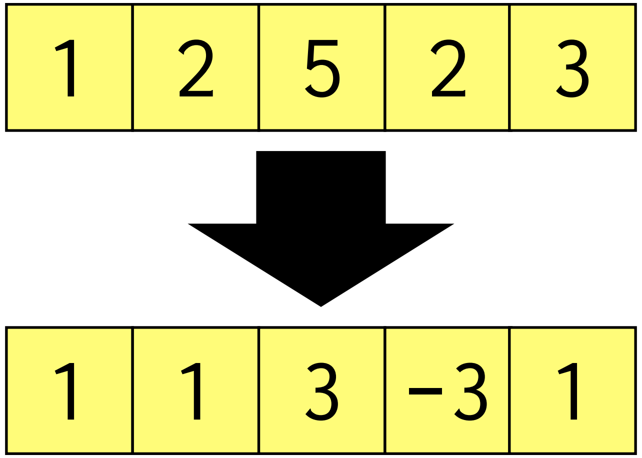
- a pre-processing for image compression
  - cf. PNG

PIP Here

Unidirectional Implementation

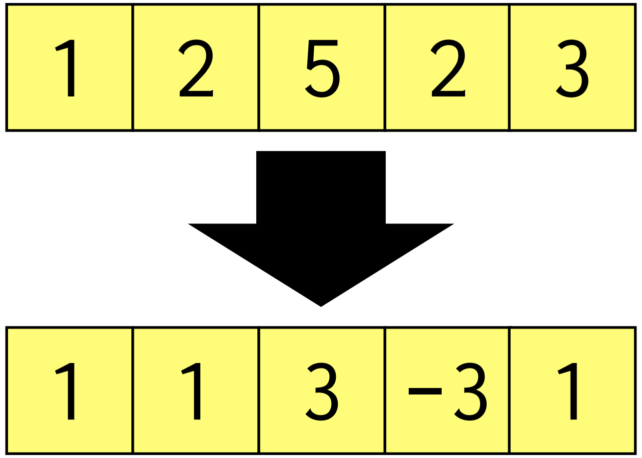
```haskell
subs :: [Int] -> [Int]
subs xs = go 0 xs

go :: Int -> [Int] -> [Int]
go n []     = []
go n (x:xs) = (x - n) : go x xs
```



PIP Here

Observations



- subs itself is invertible
- go is not invertible but *partially-invertible*
  - go n is invertible for any fixed (or, static) n

Challenge: dynamic data flow into a static position

```
subs :: [Int] -> [Int]
subs xs = go 0 xs
go :: Int -> [Int] -> [Int]
go n []       = []
go n (x:xs) = (x - n) : go x xs
```

PIP Here

Our Approach: Sparcl (1/2)

- A language for partially invertible computation, with
  - ***linear types*** (based on $\lambda^{q}_{\to}$ [Bernardy+18])
  - ***invertible types***
    - A^{R}: A-typed data to be handled only in invertible ways
    - ***invertible functions as ordinary functions***: $A^{\mathsf{R}} \multimap B^{\mathsf{R}}$
      - a unified f/w for invertible and partially invertible functions

PIP Here

```
subs  : [Int]ᴿ ⊸ [Int]ᴿ
go    : Int -> [Int]ᴿ ⊸ [Int]ᴿ
```

Our Approach: SPARCL (2/2)

- ...
- pin ***to bridge the invertible & ordinary worlds***

$$\texttt{pin} : A^{R} \multimap (A \to B^{R}) \multimap (A \otimes B)^{R}$$

- locally converts invertible values to ordinary ones
  - inspired by [Kennedy&Vytiniotis 12]

PIP Here

Handling A^R-typed Values in SPARCL

```
data Nat = Z | S Nat

add : Nat -> Natᴿ ⊸ Natᴿ
add Z     y = y
add (S x) y = Sᴿ (add x y)

mul : Nat -> Natᴿ ⊸ Natᴿ
mul x Zᴿ      = Zᴿ with isZ
mul x (S y)ᴿ = add x (mul x y)
               with not . isZ
```

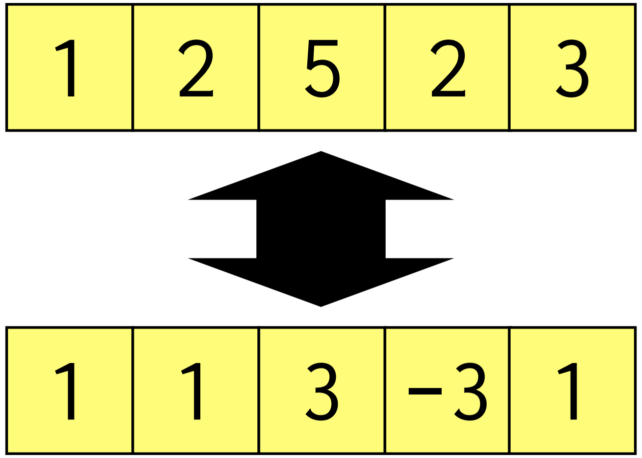
lifted constructor

$S^{R} : Nat^{R} \multimap Nat^{R}$

invertible branching [Lutz 86, Yokoyama+08]

PIP Here

subs in SPARCL




```
subs : [Int]^R ⊸ [Int]^R
subs xs = go 0 xs

go : Int -> [Int]^R ⊸ [Int]^R
go n Nil^R              = Nil^R with null
go n (Cons x xs)^R =
  let (x,r)^R = pin x (λz.go z xs)
  in Cons^R (sub n x) r with not . null

sub : Int -> Int^R ⊸ Int^R
```

cf. **pin** : $A^R \multimap (A \to B^R) \multimap (A \otimes B)^R$

PIP Here

subs in SPARCL

```
subs : [Int]^R ⊸ [Int]^R
subs xs = go 0 xs

go : Int -> [Int]^R ⊸ [Int]^R
go n Nil^R              = Nil^R with null
go n (Cons x xs)^R =
  let (x,r)^R = pin x (λz.go z xs)
  in Cons^R (sub n x) r with not . null

sub : Int -> Int^R ⊸ Int^R
```

```
cf. pin : A^R ⊸ (A -> B^R) ⊸ (A ⊗ B)^R
```

unidir. ver.

```haskell
subs :: [Int] -> [Int]
subs xs = go 0 xs

go :: Int -> [Int] -> [Int]
go n []       = []
go n (x:xs) = (x - n) : go x xs
```

PIP Here

subs in SPARCL

```
subs :: [Int] -> [Int]
subs xs = go 0 xs

go :: Int -> [Int] -> [Int]
go n []       = []
go n (x:xs) = (x - n) : go x xs
```

unidir. ver.

```
subs : [Int]ᴿ ⊸ [Int]ᴿ
subs xs = go 0 xs

go : Int -> [Int]ᴿ ⊸ [Int]ᴿ
go n Nilᴿ              = Nilᴿ with null
go n (Cons x xs)ᴿ =
  let (x,r)ᴿ = pin x (λz.go z xs)
  in Consᴿ (sub n x) r with not . null

sub : Int -> Intᴿ ⊸ Intᴿ
```

pin converts $x:\mathrm{Int}^{\mathrm{R}}$ to $z:\mathrm{Int}$

cf. **pin** : $A^{\mathrm{R}} \multimap (A \to B^{\mathrm{R}}) \multimap (A \otimes B)^{\mathrm{R}}$

PIP Here

Executing Invertible subs in SPARCL

```
> fwd subs [1,2,5,2,3]
[1,1,3,-3,1]
> bwd subs [1,1,3,-3,1]
[1,2,5,2,3]
```

fwd : $(A^R \multimap B^R)$ -> A -> B
bwd : $(A^R \multimap B^R)$ -> B -> A

PIP Here

Our Paper Includes ...

- Core system $\lambda_{\rightarrow}^{\mathrm{PI}}$ of SPARCL
  - based on a linear calculus $\lambda_{\rightarrow}^{q}$ [Bernardy+18]
  - inspired by two-staged languages [Moggi 98,...]
- Formal Properties
  - type safety & bijectivity
- Larger examples
  - Huffman encoding
  - Tree rebuilding by program calculation

PIP Here

Related Work

- Inversion methods
  - Partial inversion
    [Nishida+ 05, Almendros-Jiménez & Vidal 06]
  - Semi inversion [Mogensen 05]
- Reversible languages with limited partial invertibility
  - reversible updates [Lutz 86,...]
  - CoreFun [Jacobsen+18]

PIP Here

Related Work

- HOBiT [M&W 18]
  - a higher-order bidirectional programming language
    - lenses [Foster+05] as ordinary functions
      - `Lens S T` is represented as **`B`** `S ->` **`B`** `T`
  - not with linear types or the pin operator

PIP Here

Conclusion

- SPARCL: a programming language …
  - for writing ***invertible*** functions
  - through composing ***partially-invertible*** functions

more natural and more expressive

  - supported by ***linear types***
    - key to ***correctness by construction***

PIP Here

More Info on Implementation

`https://bx-lang.github.io/EXHIBIT/sparcl.html`