SPARCL
A Language for Partially Invertible Computation

Kazutaka Matsuda
Tohoku University, Japan

Meng Wang
University of Bristol, UK

University of BRISTOL

PPL 2021用
少し長いバージョン
presented at ICFP 2020

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

PPL 2021用
少し長いバージョン
presented at ICFP 2020

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?***

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?

Our Answer: Sparcl

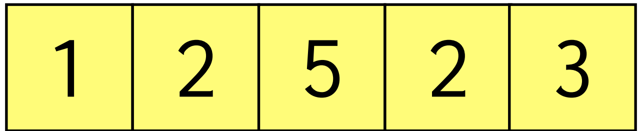
- A programming language ...
  - for writing ***invertible*** functions
  - through composing ***partially-invertible*** functions

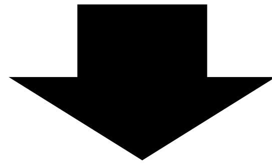
    more natural and more expressive

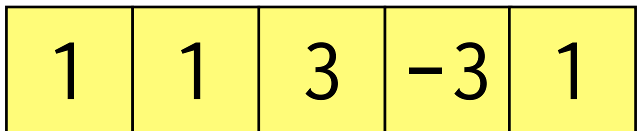
  - supported by ***linear types***
    - a key to ***correctness by construction***

Running Example

- Differences of adjacent elements in a list





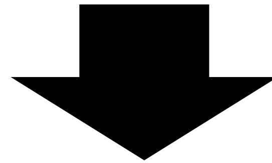


- a pre-processing for image compression
  - cf. PNG

Running Example

- Differences of adjacent elements in a list

0 | 1 | 2 | 5 | 2 | 3



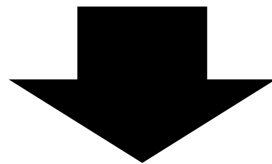
1 | 1 | 3 | -3 | 1

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

Unidirectional Implementation

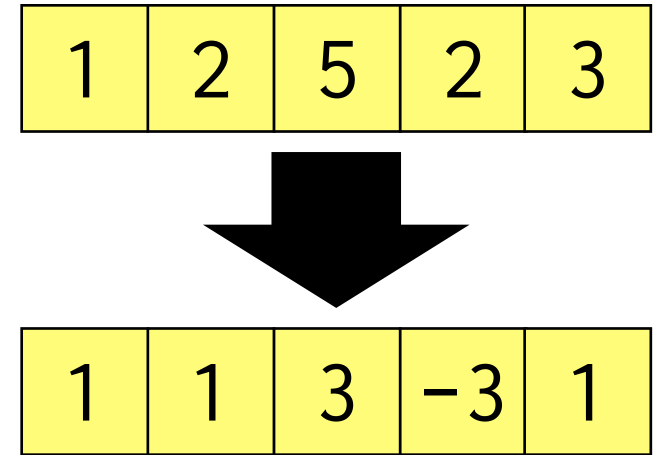
```haskell
subs :: [Int] -> [Int]
subs xs = go 0 xs

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

1	2	5	2	3



1	1	3	-3	1

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
```

Our Approach: SPARCL (1/2)

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

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

Our Approach: SPARCL (2/2)

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

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

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

Handling A^R-typed Values in SPARCL

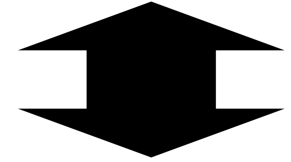
```
data Nat = Z | S Nat

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]

subs in SPARCL

1	2	5	2	3



1	1	3	-3	1

```
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** : $A^R \multimap (A \to B^R) \multimap (A \otimes B)^R$

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
```

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

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 : IntR to z : Int

cf. **pin** : AR ⊸ (A -> BR) ⊸ (A ⊗ B)R

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 : (Aᴿ ⊸ Bᴿ) -> A -> B
bwd : (Aᴿ ⊸ Bᴿ) -> B -> A
```

A Teaser of Technicality

- Core system λ^{PI} of Sparcl
  - syntax & semantics
- Properties

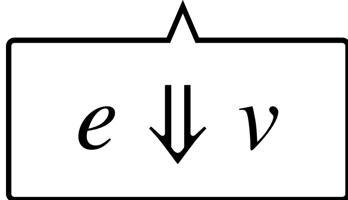
Core System: $\lambda_{\rightarrow}^{\text{PI}}$

```
e ::= x | λx.e | e₁ e₂
    | C e₁ … eₙ  | case e of { pᵢ -> eᵢ }
    | Cᴿ e₁ … eₙ | case e of { pᵢᴿ -> eᵢ with eᵢ'}
    | pin e₁ e₂
    | unlift e | e₁ ▷ e₂ | e₁ ◁ e₂
```

```
unlift : (Aᴿ ⊸ Bᴿ) -> Bij A B
(▷) : Bij A B ⊸ A -> B
(◁) : Bij A B ⊸ B -> A
```

Semantics

- ***Unidirectional***, then ***forward*** and ***backward*** [M&W 18]



$$(\lambda f \,.\, f\ (f\ x))\ (\lambda z \,.\, \mathsf{S}^{\mathrm{R}}\ z) \Downarrow \mathsf{S}^{\mathrm{R}}\ (\mathsf{S}\ x)$$

Semantics

- ***Unidirectional***, then ***forward*** and ***backward*** [M&W 18]

$e \Downarrow v$

$$(\lambda f\,.\,f\,(f\,x))\;(\lambda z\,.\,\mathsf{S}^{\mathrm{R}}\;z) \Downarrow \mathsf{S}^{\mathrm{R}}\;(\mathsf{S}\;x)$$

$v ::= \cdots \mid \langle x\,.\,E\rangle \mid E$

$E ::= x \mid \mathsf{C}^{\mathbf{R}}\;\overline{E} \mid \mathbf{case}\;E\;\mathbf{of}\;\{p_i{}^{\mathbf{R}} \to e_i\;\mathbf{with}\;\lambda x_i\,.\,e_i\}_i \mid \mathbf{pin}\;E\;(\lambda x\,.\,e)$

Semantics

- ***Unidirectional***, then ***forward*** and ***backward*** [M&W 18]

$e \Downarrow v$ $\quad\mu \vdash E \Rightarrow v$ $\quad E \Leftarrow v \dashv \mu$

$$(\lambda f\,.\,f\,(f\,x))\;(\lambda z\,.\,\mathsf{S}^{\mathrm{R}}\;z) \Downarrow \mathsf{S}^{\mathrm{R}}\;(\mathsf{S}\;x)$$

ready for ***forward*** & ***backward*** eval.

$\{x = \mathsf{Z}\} \vdash \mathsf{S}^{\mathbf{R}}\;(\mathsf{S}^{\mathbf{R}}\;x) \Rightarrow \mathsf{S}\;(\mathsf{S}\;\mathsf{Z})$ $\qquad \mathsf{S}^{\mathbf{R}}\;(\mathsf{S}^{\mathbf{R}}\;x) \Leftarrow \mathsf{S}\;(\mathsf{S}\;\mathsf{Z}) \dashv \{x = \mathsf{Z}\}$

$$v ::= \cdots \mid \langle x\,.\,E\rangle \mid E$$

$$E ::= x \mid \mathsf{C}^{\mathbf{R}}\;\overline{E} \mid \mathbf{case}\;E\;\mathbf{of}\;\{p_i^{\mathbf{R}} \to e_i\;\mathbf{with}\;\lambda x_i\,.\,e_i\}_i \mid \mathbf{pin}\;E\;(\lambda x\,.\,e)$$

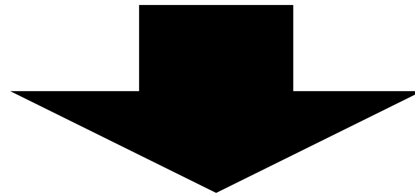
Properties

Subject Reduction

$$\varnothing; \Theta \vdash e : A \text{ and } e \Downarrow v \text{ implies } \varnothing; \Theta \vdash v : A$$

Bijectivity of Residuals

$$\mu \vdash E \Rightarrow v \text{ if and only if } E \Leftarrow v \dashv \mu$$



Suppose $\varnothing; \varnothing \vdash e : \text{Bij}\ A\ B$. Then, $e \triangleright v \Downarrow v'$ iff $e \triangleleft v' \Downarrow v$

e must evaluate to $\langle x \,.\, E \rangle$

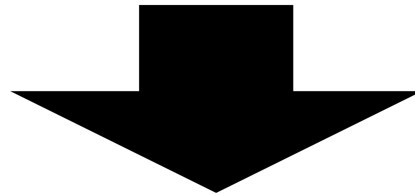
Properties

Subject Reduction

$\varnothing;\Theta \vdash e : A$ and $e \Downarrow v$ implies $\varnothing;\Theta \vdash v : A$

Bijectivity of Residuals

$\mu \vdash E \Rightarrow v$ if and only if $E \Leftarrow v \dashv \mu$



Suppose $\varnothing;\varnothing \vdash e$: Bij A B. Then, $e \triangleright v \Downarrow v'$ iff $e \triangleleft v' \Downarrow v$

e must evaluate to $\langle x \,.\, E \rangle$

Properties

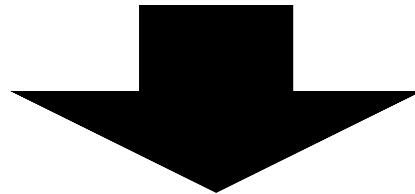
$$\frac{\{x = v\} \vdash E \Rightarrow v'}{\langle x \,.\, E \rangle \triangleright v \Downarrow v'} \qquad \frac{E \Leftarrow v' \dashv \{x = v\}}{\langle x \,.\, E \rangle \triangleleft v' \Downarrow v}$$

Subject Reduction

$\emptyset; \Theta \vdash e : A$ and $e \Downarrow v$ implies $\emptyset; \Theta \vdash v : A$

Bijectivity of Residuals

$\mu \vdash E \Rightarrow v$ if and only if $E \Leftarrow v \dashv \mu$



Suppose $\emptyset; \emptyset \vdash e :$ Bij $A\ B$. Then, $e \triangleright v \Downarrow v'$ iff $e \triangleleft v' \Downarrow v$

e must evaluate to $\langle x \,.\, E \rangle$

Our Paper Includes ...

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

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]

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

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***

More Info on Implementation

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

huff in SPARCL

```
huff : [Symbol]ᴿ ⊸ (HuffTable ⊗ [Bit])ᴿ
huff s =
  let (s,h)ᴿ = pin s (λs'.new eqHuff (makeHuff s'))
  in pin h (λh'. encode h' s)
```

```
new      : (a -> a -> Bool) -> a -> aᴿ
makeHuff : [Symbol] -> HuffTable
encode   : HuffTable -> [Symbol]ᴿ ⊸ [Bit]ᴿ
```

huff in SPARCL

```
huff : [Symbol]ᴿ ⊸ (HuffTable ⊗ [Bit])ᴿ
huff s =                              : [Symbol]
  let (s,h)ᴿ = pin s (λs'.new eqHuff (makeHuff s'))
  in pin h (λh'. encode h' s)
```

```
new      : (a -> a -> Bool) -> a -> aᴿ
makeHuff : [Symbol] -> HuffTable
encode   : HuffTable -> [Symbol]ᴿ ⊸ [Bit]ᴿ
```

huff in SPARCL

```
huff : [Symbol]ᴿ ⊸ (HuffTable ⊗ [Bit])ᴿ
huff s =  : HuffTableᴿ           : [Symbol]
  let (s,h)ᴿ = pin s (λs'.new eqHuff (makeHuff s'))
  in pin h (λh'. encode h' s)
```

```
new      : (a -> a -> Bool) -> a -> aᴿ
makeHuff : [Symbol] -> HuffTable
encode   : HuffTable -> [Symbol]ᴿ ⊸ [Bit]ᴿ
```

huff in SPARCL

```
huff : [Symbol]ᴿ ⊸ (HuffTable ⊗ [Bit])ᴿ
huff s =     : HuffTableᴿ              : [Symbol]
  let (s,h)ᴿ = pin s (λs'.new eqHuff (makeHuff s'))
  in pin h (λh'. encode h' s)
                 : HuffTable

new      : (a -> a -> Bool) -> a -> aᴿ
makeHuff : [Symbol] -> HuffTable
encode   : HuffTable -> [Symbol]ᴿ ⊸ [Bit]ᴿ
```