

Program Inversion and Reversible Computation 2020

*Reversible Computing:
Janus (1)*

A Reversible Programming Language

Tetsuo Yokoyama
(revised by Robert Glück)

Today's Plan

- Review: Fibonacci Pairs Program
- Formalization of Janus
 - Syntax
 - Semantics of expressions and statements
- Reversibility of Janus
- Program inverter for Janus
- Exercise

2



Review: Fibonacci Pairs

Function fib: the n -th pair of Fibonacci numbers:

```
fib(0) = [1 1]
fib(1) = [1 2]
fib(8) = [34 55]
```

[1 1]	2 3 5 ...	[34 55] ...
0-th pair		
8-th pair		

Inverse function fib⁻¹: the index of a Fibonacci pair:

```
fib-1([1 1]) = 0
fib-1([1 2]) = 1
fib-1([34 55]) = 8
```

[GlückKawabe03]



Example: Fibonacci-Pairs

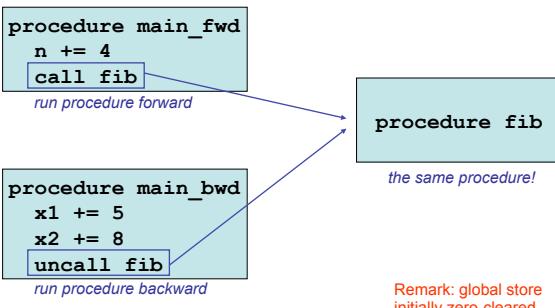
n x1 x2	global store (three integer variables)
procedure fib	
if n=0 then x1 += 1	reversible
test x2 += 1	procedure
else n -= 1	
call fib	
x1 += x2	
x1 <=> x2	
fi x1=x2	swap values of x_1, x_2
	assertion

3

4



Forward & Backward Computation



5



Janus: a Reversible Language

- Imperative language
- Global store, no local store
- Scalar and array types, only integer values
- Structured control operators (IF, LOOP)
- Simple recursive procedures
 - No return value, no local variables, side effects on global store
- Procedures can be invoked in two ways: **call** and **uncall**
- Without information loss: injective partial functions
- History-free computation

6



Syntax of Janus

```

 $p ::= vdec^* \text{ (procedure } id \text{ } s)^+$  Assignment operations
 $vdec ::= x \mid x[c]$ 
 $s ::= [x \oplus= e \mid x[e] \oplus= e]$  Reversible Conditional
 $\text{if } e \text{ then } s \text{ else } s \text{ fi } e$  Reversible Loop
 $\text{from } e \text{ do } s \text{ loop } s \text{ until } e$ 
 $\text{call } id \mid \text{uncall } id \mid \text{skip} \mid s \text{ s}$  Procedure call/uncall
 $e ::= c \mid x \mid x[e] \mid \sim e \mid e \odot e$ 
 $c ::= 0 \mid 1 \mid \dots \mid 4294967295$  32-bits integers
 $\oplus ::= + \mid - \mid \sim$ 
 $\odot ::= \oplus \mid * \mid / \mid \% \mid */ \mid \& \mid \mid \mid \&& \mid \mid \mid$ 
 $< \mid > \mid = \mid != \mid \leq \mid \geq$ 

```

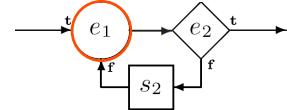
7



Reversible Loop Structured Control Flow

Flowchart diagram:

NEW: assertion at entry



Textual representation:

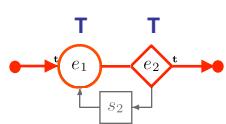
FROM e_1 **LOOP** s_2 **UNTIL** e_2

8

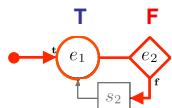


Execution of Reversible Loop

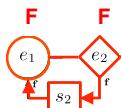
Skip:



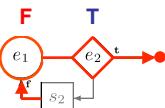
1. Enter loop:



2. Repeat:



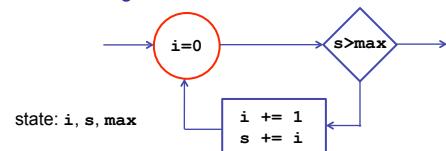
3. Exit loop:



9

Example: Reversible While Loop

Flowchart diagram:



Textual representation:

```

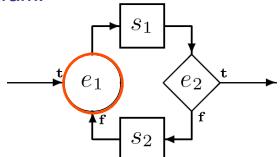
FROM  $i=0$ 
LOOP  $i += 1$ 
 $s += i$ 
UNTIL  $s > \text{max}$ 

```

10

General Reversible Loop

Flowchart diagram:

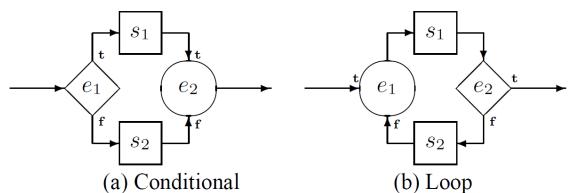
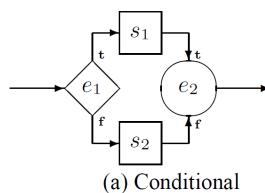


Programming language:

FROM e_1 **DO** s_1 **LOOP** s_2 **UNTIL** e_2

11

Reversible Structured Control Flow



if e_1 *then* s_1 *else* s_2 *fi* e_2 *from* e_1 *do* s_1 *loop* s_2 *until* e_2

12

Examples

Reversible conditional

```
procedure fib
  IF n=0 THEN x1 += 1
    x2 += 1
  ELSE n -= 1
    CALL fib
    x1 += x2
    x1 <=> x2
  FI x1=x2
```

Reversible loop

```
procedure sum
  FROM i=0
  DO i += 1
  LOOP s += i
  UNTIL s>max
```

Reversible C-like Assignments

Addition

 $x += e$


Subtraction

 $x -= e$


Bitwise XOR

 $x ^= e$


Local inversion

Rule: variable x must not occur in expression e .

Abbreviation: $x \oplus= e \Leftrightarrow x := x \oplus e$

13

14

What is Reversibility?

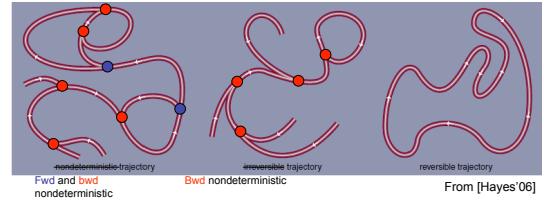
Reversible	\Leftrightarrow	Locally invertible
	\Leftrightarrow	Fwd and bwd deterministic

Locally invertible:

- Given any part composed of 'atomic units' of a program, its inverse can be constructed without global analysis.

• We will show that Janus statements are reversible.

Review: Reversible Trajectory



- A reversible computing system has at any time at most a single previous computation state as well as a single next computation state.
- Unique choice in both directions.

15

16

Typical Sources of Bwd Non-determinism

- Control Flow**
 - Conditional: $\text{if } ... \text{ then } ... \text{ else } ... \text{ end}$
 - Loop: $\text{for } (i=0; i < n; i += 3) \{ ... \}$
 - Tail-Recursion: $f \ x \stackrel{\text{def}}{=} \text{if } ... \text{ then } ... \text{ else } f(x-1)$
- Primitive Operations**
 - Destructive assignment: $x := 3$
 - Non-injective assignment: $x -= x$

Those constructs are not available in Janus.

Evaluation of Expressions

Judgment:	$\sigma \vdash_{\text{expr}} e \Rightarrow v$
Store	Exp
Exp	Val

$\sigma \vdash_{\text{expr}} c \Rightarrow \llbracket c \rrbracket$ Con $\sigma \vdash_{\text{expr}} x \Rightarrow \sigma(x)$ Var

$\frac{\sigma \vdash_{\text{expr}} e_1 \Rightarrow v_1 \quad \sigma \vdash_{\text{expr}} e_2 \Rightarrow v_2 \quad \llbracket \odot \rrbracket(v_1, v_2) = v}{\sigma \vdash_{\text{expr}} e_1 \odot e_2 \Rightarrow v}$ BinOp

Store $\sigma : \text{Var} \Rightarrow \text{Val}$

17

18

Non-injective Binary Operators

- Some of the binary operators (others are similar):

$$\llbracket + \rrbracket(v_1, v_2) = (v_1 + v_2) \bmod 2^{32}$$

$$\llbracket = \rrbracket(v_1, v_2) = \begin{cases} 0 & \text{if } v_1 \neq v_2 \\ 1 & \text{if } v_1 = v_2 \end{cases}$$

- None of these binary operators is injective.
- No unique inverse operator exists.

Forward Determinism of Expressions

- The evaluation of expressions is forward deterministic:

$$\forall e \in \text{Exps[Janus]}, \forall \sigma \in \text{Stores[Janus]}. \quad \sigma \vdash_{\text{expr}} e \Rightarrow v' \wedge \sigma \vdash_{\text{expr}} e \Rightarrow v'' \implies v' = v''$$

- But *not backward* deterministic:

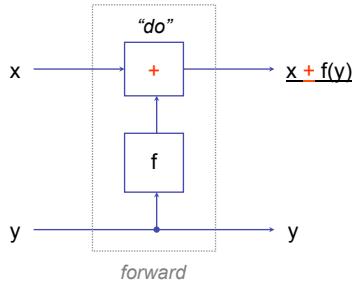
Ex.
 $\{x \mapsto 2, y \mapsto 3\} \vdash_{\text{expr}} x + y \Rightarrow 5$
 $\{x \mapsto 1, y \mapsto 4\} \vdash_{\text{expr}} x + y \Rightarrow 5$

Why does this *not* harm
the reversibility of statements?

19

20

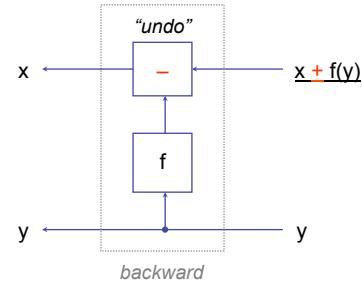
Reversible Update



+ ... injective in x-argument

21

Reversible Update



+ ... injective in x-argument

22

Injective in 1st Argument

If an operator \odot is **injective in its 1st argument**,
 $a \odot c = b \odot c \Rightarrow a = b$ (if $a \odot c$ and $b \odot c$ defined),
then an operator \odot exists to **reconstruct 1st argument**:
 $(a \odot c) \odot c = a$

Example:

$$n+3 = m+3 \Rightarrow n = m \text{ and } (n+3)-3 = n$$

Non-Example:

$$n^0 = m^0 \not\Rightarrow n = m$$

23

Reversible Update

Given (\odot, \odot) and a partial function f , then function
 $g(x, y) = (x \odot f(y), y)$
is a **reversible update of x**, and there exists
 $g^{-1}(x, y) = (x \odot f(y), y)$

Example:

$$g(x, y) = (x + f(y), y) \dots \text{reversible update of } x$$

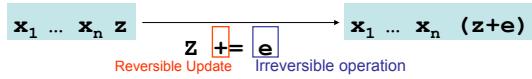
$$g^{-1}(x, y) = (x - f(y), y) \dots \text{its inverse function}$$

Fct f can be non-injective; x, y can be tuples $(x_1, \dots, x_m), (y_1, \dots, y_m)$.
Every reversible update g is an injective function.

24

Examples: Reversible Updates

- Janus assignments:



- Toffoli gate:

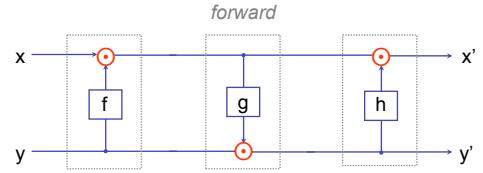
$$TG(x, y, z) = (x, y, z \wedge (x \& y))$$

Reversible Update Irreversible operation

- Reversible updates are also used as instructions in the reversible assembly language PISA.

25

Composition of Reversible Updates



sequence of reversible updates
in any combination is reversible

26

Reversible C-like Assignments

$$\begin{array}{c} \text{Evaluation of RHS} \\ (\text{Irreversible}) \end{array} \quad \boxed{\sigma \vdash \text{expr } e \Rightarrow v} \quad \boxed{\text{Reversible Update}} \quad \boxed{v_2 = [\oplus](v_1, v)} \quad \text{AssVar}$$

$$\sigma \uplus \{x \mapsto v_1\} \vdash_{stmt} x \oplus= e \Rightarrow \sigma \uplus \{x \mapsto v_2\}$$

- $\sigma \vdash \text{expr } e \Rightarrow v$ is fwd deterministic.
- Variable x must not occur in expression e .
- Function $\lambda v'. [\oplus](v', v)$ is injective for any v when \oplus is $+$, $-$ or \wedge .

\Rightarrow It has an inverse function.

Statement $x \oplus= e$ is reversible.

27

Reversible Conditional

$$\begin{array}{c} \text{Test} \\ \boxed{\sigma \vdash \text{expr } e_1 \Rightarrow v_1} \quad \boxed{\sigma \vdash \text{stmt } s_1 \Rightarrow \sigma'} \\ \text{is-true?}(v_1) \end{array} \quad \begin{array}{c} \text{Assertion} \\ \boxed{\sigma' \vdash \text{expr } e_2 \Rightarrow v_2} \\ \text{is-true?}(v_2) \end{array} \quad \text{IfTrue}$$

$$\sigma \vdash_{stmt} \text{if } e_1 \text{ then } s_1 \text{ else } s_2 \text{ fi } e_2 \Rightarrow \sigma'$$

- $\sigma \vdash \text{expr } e_1 \Rightarrow v_1$ and $\sigma \vdash \text{expr } e_2 \Rightarrow v_2$ are fwd deterministic.
- We assume that $\sigma \vdash \text{stmt } s_1 \Rightarrow \sigma'$ is reversible.

\Rightarrow IfTrue is reversible.

(IfFalse is similar, and thus the conditional is reversible.)

28

Reversible Loop

- Similar to reversible conditional.

$$\begin{array}{c} \text{Assertion} \\ \boxed{\sigma \vdash \text{expr } e_1 \Rightarrow v_1} \quad \boxed{\sigma \vdash \text{loop1 } (e_1, s_1, s_2, e_2) \Rightarrow \sigma'} \\ \text{is-true?}(v_1) \end{array} \quad \begin{array}{c} \text{Test} \\ \boxed{\sigma' \vdash \text{expr } e_2 \Rightarrow v_2} \\ \text{is-true?}(v_2) \end{array}$$

$$\sigma \vdash_{stmt} \text{from } e_1 \text{ do } s_1 \text{ loop } s_2 \text{ until } e_2 \Rightarrow \sigma'$$

LoopMain

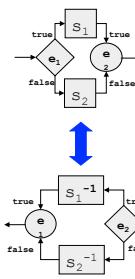
- $\sigma \vdash \text{expr } e_1 \Rightarrow v_1$ and $\sigma \vdash \text{expr } e_2 \Rightarrow v_2$ are fwd deterministic.
- We assume that $\sigma \vdash \text{loop1 } (e_1, s_1, s_2, e_2) \Rightarrow \sigma'$ is reversible.

\Rightarrow LoopMain is reversible.

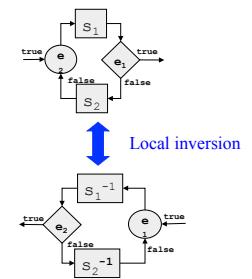
29

Local Inversion of Reversible CFOs

Conditional (IF)



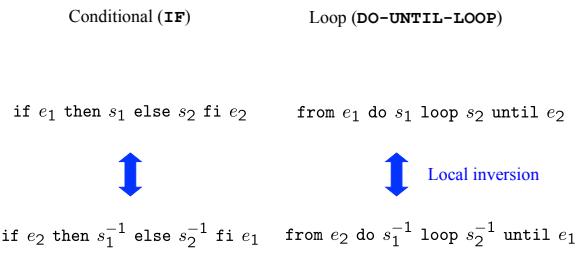
Loop (DO-UNTIL-LOOP)



Local inversion

30

Local Inversion of Reversible CFOs



31

Procedure Call / Uncall

$$\frac{\boxed{\sigma} \vdash_{stmt} \Gamma(id) \Rightarrow \boxed{\sigma'}}{\sigma \vdash_{stmt} \text{call } id \Rightarrow \sigma'} \text{ Call}$$

$$\frac{\boxed{\sigma} \vdash_{stmt} \text{uncall } id \Rightarrow \sigma'}{\sigma \vdash_{stmt} \text{uncall } id \Rightarrow \sigma'} \text{ Uncall}$$

$$\Gamma \in \text{Idens[Janus]} \rightarrow \text{Stmts[Janus]}$$

- Procedure call / uncall is reversible.

↑
call id
↓
uncall id

↔ Local inversion

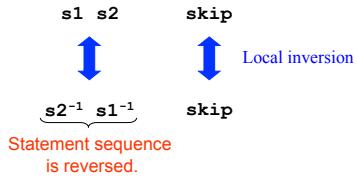
32

Skip and Sequence

$$\frac{\sigma \vdash_{stmt} \text{skip} \Rightarrow \sigma}{\sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \quad \sigma' \vdash_{stmt} s_2 \Rightarrow \sigma''} \text{ Seq}$$

$$\sigma \vdash_{stmt} s_1 s_2 \Rightarrow \sigma''$$

- Skip and sequence are reversible.



33

Short Summary: Reversibility of Janus

- Janus statements are reversible.

$$\boxed{\forall s \in \text{Stmts[Janus]}, \exists s' \in \text{Stmts[Janus]}, \forall \sigma, \sigma' \in \text{Stores[Janus]}. \sigma \vdash_{stmt} s \Rightarrow \sigma' \iff \sigma' \vdash_{stmt} s' \Rightarrow \sigma}}$$

(※ Evaluation of expressions is not reversible.
This does not harm reversibility.)

34

Computing Strength of Janus

- Any irreversible computation can be simulated by reversible computation:

Adding history tape + clearing history tape
(Landauer embedding + Bennett's method)
- Reversible Turing-machine interpreter can be implemented in Janus without using a history.

A reversible Turing machine is injective.

35

Properties of Expressions and Statements

• Equivalence Relations

$$e_1 \sim e_2 \text{ iff } (\forall v \in \text{Vals[Janus]}, \forall \sigma \in \text{Stores[Janus]}. \sigma \vdash_{expr} e_1 \Rightarrow v \iff \sigma \vdash_{expr} e_2 \Rightarrow v)$$

$$s_1 \sim s_2 \text{ iff } (\forall \sigma, \sigma' \in \text{Stores[Janus]}. \sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \iff \sigma \vdash_{stmt} s_2 \Rightarrow \sigma')$$

• Inverse

- We write an inverse of s as s^{-1} .
- Inverse function is unique up to equivalence relation.

36

Properties of Statements

- Inverse

– Inverse statements are commutative

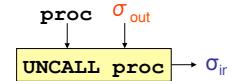
$$s_1 \ s_2 \sim \epsilon \iff s_2 \ s_1 \sim \epsilon$$

- Sequential inverse

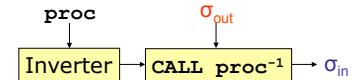
$$(s_1 \ s_2)^{-1} \sim s_2^{-1} \ s_1^{-1}$$

Review: Two Approaches to Inversion of Programs

- Inverse Interpretation:



- Program Inversion:



37

38

Example: Program Inversion

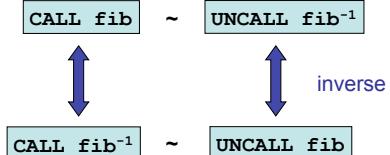
`n x1 x2` global store (three integer variables)

```

procedure fib
  if n=0 then x1+=1
  test      x2+=1
  else n-=1
    call fib
    x1+=x2
    x1<=>x2
  fi [x1=x2] assertion
  
```

Program Inversion \mathcal{I}

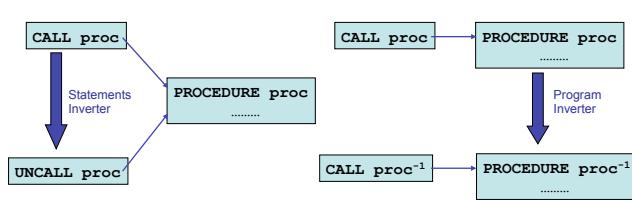
Two Ways of Invoking Procedures



39

40

Two Ways of Inversion



41

Statement Inverter

$$\begin{aligned}
 \mathcal{I}[x \oplus e] &= x \oplus' e \quad \text{where } (\oplus') = \mathcal{I}_{op}[\oplus] \\
 \mathcal{I}[x[e_i] \oplus e] &= x[e_i] \oplus' e \quad \text{where } (\oplus') = \mathcal{I}_{op}[\oplus] \\
 \mathcal{I}[\text{if } e_1 \text{ then } s_1 \text{ } \square] &= \text{if } e_2 \text{ then } \mathcal{I}[s_1] \\
 \mathcal{I}[\text{else } s_2 \text{ fi } e_2 \text{ } \square] &= \text{else } \mathcal{I}[s_2] \text{ fi } e_1 \\
 \mathcal{I}[\text{from } e_1 \text{ do } s_1 \text{ } \square] &= \text{from } e_2 \text{ do } \mathcal{I}[s_1] \\
 \mathcal{I}[\text{loop } s_2 \text{ until } e_2 \text{ } \square] &= \text{loop } \mathcal{I}[s_2] \text{ until } e_1 \\
 \mathcal{I}[\text{call id}] &= \text{uncall id} \\
 \mathcal{I}[\text{uncall id}] &= \text{call id} \\
 \mathcal{I}[\text{skip}] &= \text{skip} \\
 \mathcal{I}[s_1 \ s_2] &= \mathcal{I}[s_2] \ \mathcal{I}[s_1]
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{I}_{op}[\oplus] &= - \\
 \mathcal{I}_{op}[\ominus] &= + \\
 \mathcal{I}_{op}[\wedge] &= \wedge
 \end{aligned}$$

42

Statement Inverter

$$\forall s \in \text{Stmts[Janus]}, \exists s^{-1} \in \text{Stmts[Janus]}. s^{-1} \sim \mathcal{I}[s]$$

Ex.



Statement Inverter

43

Program Inverter

$$\begin{aligned} \mathcal{I}[vdec^* proc_1 \dots proc_n] &= vdec^* \mathcal{I}[proc_1] \dots \mathcal{I}[proc_n] \\ \mathcal{I}[\text{procedure } id\ s] &= \text{procedure } id^{-1}\ \mathcal{I}[s] \\ \mathcal{I}[x \oplus= e] &= x \oplus' = e \quad \text{where } (\oplus') = \mathcal{I}_{op}[\oplus] \\ \mathcal{I}[x[e_1] \oplus= e] &= x[e_1] \oplus' = e \quad \text{where } (\oplus') = \mathcal{I}_{op}[\oplus] \\ \mathcal{I}[\text{if } e_1 \text{ then } s_1 \text{ else } s_2 \text{ fi } e_2] &= \text{if } e_2 \text{ then } \mathcal{I}[s_1] \text{ else } \mathcal{I}[s_2] \text{ fi } e_1 \\ \mathcal{I}[\text{from } e_1 \text{ do } s_1 \text{ loop } s_2 \text{ until } e_2] &= \text{from } e_2 \text{ do } \mathcal{I}[s_1] \text{ loop } \mathcal{I}[s_2] \text{ until } e_1 \\ \mathcal{I}[\text{call } id] &= \text{call } id^{-1} \\ \mathcal{I}[\text{uncall } id] &= \text{uncall } id^{-1} \\ \mathcal{I}[\text{skip}] &= \text{skip} \\ \mathcal{I}[s_1\ s_2] &= \mathcal{I}[s_2]\ \mathcal{I}[s_1] \\ \mathcal{I}_{op}[\cdot^+ &= - \\ \mathcal{I}_{op}[\cdot^- &= + \\ \mathcal{I}_{op}[\cdot^\wedge &= ^\wedge \end{aligned}$$

44

Properties of Program Inverter

- Program inversion is performed by a recursive decent over each procedure.
 - A program inverter is inverse to itself.
- $$s \sim \mathcal{I}[\mathcal{I}[s]]$$
- The length of a program is unchanged.
 - The length of computation is unchanged.
 - $\text{call fib} \sim \text{uncall fib}^{-1}$

45

Janus Playground

<http://topps.diku.dk/pirc/janus-playground/>

The screenshot shows a window titled "Janus Playground" with a toolbar at the top. The main area contains a code editor with the following pseudocode:

```

1 // Fibonacci Pairs
2
3 n x1 x2
4
5 procedure fib
6 if n=0 then x1 += 1
7 | x2 += 1
8 | else n -= 1
9 | call fib
10 | x1 += x2
11 | x1 <=> x2
12 fi x1=x2
13
14 procedure main
15 n += 4
16 call fib
17

```

Below the code editor, there is a terminal-like window showing the execution results:

```

n = 0
x1 = 5
x2 = 8

```

51

Related Work: History of Reversible Programming Languages

- **Janus (Lutz and Derby 1982)**
 - The first reversible language. Imperative.
- **Ψ -Lisp (Baker 1992)**
 - The reversible Lisp-like functional language w/ destructive updates.
- **R (Frank 1997)**
 - R compiler generates PISA code, which runs on the reversible processor Pendulum.
- **Inv (Mu et al. 2004)**
 - An injective pure functional language.

54

Relation to Program Inversion

- Theoretically, inverse computation of any program is possible using McCarthy's generate-and-test method (1956).

But

- This method is too inefficient to be practical.
- There might be no unique solution.

In Janus

- Inverse computation is efficient.
- The solution is unique (injective programs only).

55

Summary: A Reversible Programming Language

- Programs implement *injective partial* functions.
- No information loss (non-destructive computing).
- The programming language consists of *locally invertible constructs* (efficient inverse computing).
- Supports *deterministic fwd and bwd computation*.
- Inversion can be done *on-the-fly* in the interpreter.
- Program inversion is easy.

Reference

Tetsuo Yokoyama, Robert Glück.

A Reversible Programming Language and its Invertible Self-Interpreter.

In: Partial Evaluation and Program Manipulation (PEPM 2007),
(Nice, France), pages 144-153, ACM Press 2007.