Program Inversion
and Reversible Computation
2025

Reversible Computing:
Janus (1)
A Reversible Programming Language

Tetsuo Yokoyama
(revised by Robert Glück)

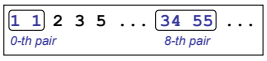
Today's Plan

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

Review: Fibonacci Pairs

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

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



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                          reversible
  if n=0 then x1 += 1                  procedure
       test  x2 += 1
     else n -= 1
          call fib
          x1 += x2
          x1 <=> x2                    swap values of x1, x2
  fi x1=x2
     assertion
```


Forward & Backward Computation

```
procedure main_fwd
  n += 4
  call fib
```
run procedure forward

```
procedure fib
```
the same procedure!

```
procedure main_bwd
  x1 += 5
  x2 += 8
  uncall fib
```
run procedure backward

Remark: global store initially zero-cleared.

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

Syntax of Janus

p ::= $vdec^*$ `(procedure` id $s)^+$
$vdec$ ::= x | x`[`c`]`
s ::= x `⊕=` e | x`[`e`]` `⊕=` e
`if` e `then` s `else` s `fi` e
`from` e `do` s `loop` s `until` e
`call` id | `uncall` id | `skip` | s s
e ::= c | x | x`[`e`]` | `~`e | e ⊙ e
c ::= `0` | `1` | ··· | `4294967295`
⊕ ::= `+` | `-` | `^`
⊙ ::= ⊕ | `*` | `/` | `%` | `*/` | `&` | `|` | `&&` | `||` | `<` | `>` | `=` | `!=` | `<=` | `>=`

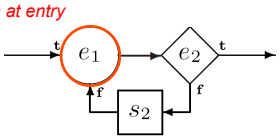
Assignment operations
Reversible Conditional
Reversible Loop
Procedure call/uncall
32-bits integers

Reversible Loop

Structured Control Flow

Flowchart diagram:

NEW: assertion at entry

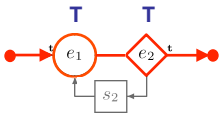


Textual representation:

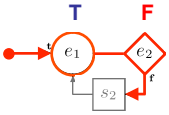
`FROM` e_1 `LOOP` s_2 `UNTIL` e_2

Execution of Reversible Loop

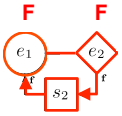
Skip:



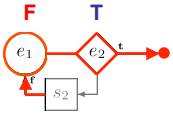
1. Enter loop:



2. Repeat:

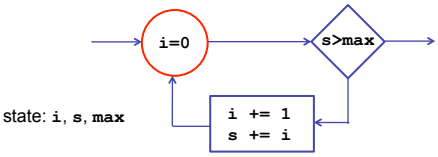


3. Exit loop:



Example: Reversible While Loop

Flowchart diagram:

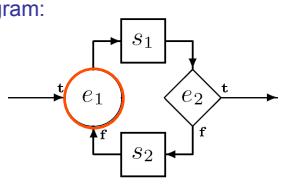


Textual representation:

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


General Reversible Loop

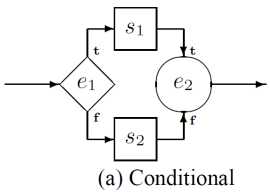
Flowchart diagram:



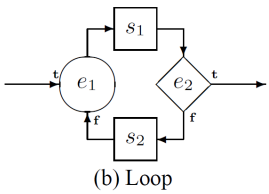
Programming language:

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

Reversible Structured Control Flow



(a) Conditional



(b) Loop

`if` e_1 `then` s_1 `else` s_2 `fi` e_2

`from` e_1 `do` s_1 `loop` s_2 `until` e_2

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	Subtraction	Bitwise XOR
`x += e`	`x -= e`	`x ^= e`
⇕	⇕	⇕ Local inversion
`x -= e`	`x += e`	`x ^= e`

Rule: variable `x` must not occur in expression `e`.

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

What is Reversibility?

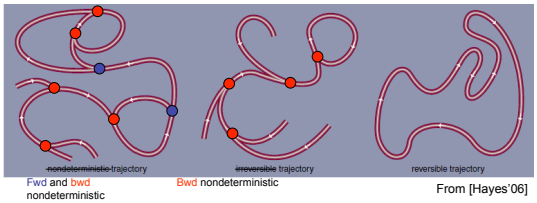
Reversible	⇔	Locally invertible
	⇔	Fwd and bwd deterministic

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



From [Hayes'06]

- 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.

Typical Sources of Bwd Non-determinism

- Control Flow
  - × Conditional: `if … then … else … end`
  - × Loop: `for (i=0; i<n; i+=3) { … }`
  - × Tail-Recursion: `f x` $\stackrel{\text{def}}{=}$ `if … then … 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: $\sigma \vdash_{expr} e \Rightarrow v$ (Store, Exp, Val)

$$\frac{}{\sigma \vdash_{expr} c \Rightarrow [\![c]\!]}\ \text{Con} \qquad \frac{}{\sigma \vdash_{expr} x \Rightarrow \sigma(x)}\ \text{Var}$$

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

Store σ : Var ⇒ Val

Non-injective Binary Operators

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

$$[\![+]\!](v_1, v_2) = (v_1 + v_2) \mod 2^{32}$$
$$[\![=]\!](v_1, v_2) = \begin{cases} 0 \ if \ v_1 \neq v_2 \\ 1 \ if \ v_1 = v_2 \end{cases}$$

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

Forward Determinism of Expressions

- The evaluation of expressions is forward deterministic:

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

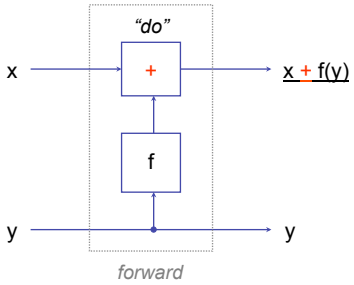
- But *not backward* deterministic:

Ex.

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

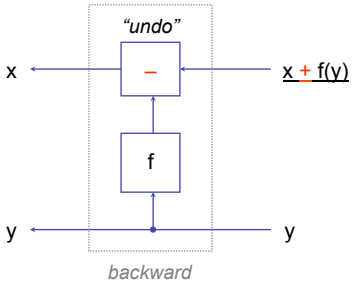
Why does this *not* harm the reversibility of statements?

Reversible Update



+ … injective in x-argument

Reversible Update



+ … injective in x-argument

Injective in 1st Argument

If an operator ⊙ is **injective in its 1st argument**,
a ⊙ **c** = b ⊙ **c** ⇒ a = b (if a⊙**c** and b⊙**c** defined),
then an operator ⊘ exists to **reconstruct 1st argument**:
(a ⊙ **c**) ⊘ **c** = a

Example:
n + **3** = m + **3** ⇒ n = m and (n + **3**) – **3** = n

Non-Example:
n * **0** = m * **0** ⇏ n = m

Reversible Update

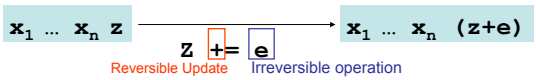
Given (⊙, ⊘) and a partial function **f**, **then** function
$g(x, y) =^{def} (x \odot \mathbf{f(y)}, y)$
is a **reversible update of x**, and there **exists**
$g^{-1}(x, y) =^{def} (x \oslash \mathbf{f(y)}, y)$

Example:
g (x, y) = (x + **f(y)**, y) … reversible update of x
g^{-1}(x, y) = (x – **f(y)**, y) … its inverse function

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

Examples: Reversible Updates

- Janus assignments:



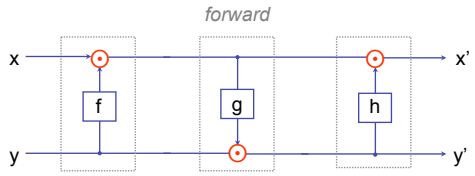
- Toffoli gate:

```
TG(x,y,z) = (x,y,z^(x&&y))
```

Reversible Update Irreversible operation

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

Composition of Reversible Updates



sequence of reversible updates
in any combination is reversible

Reversible C-like Assignments

Evaluation of RHS (Irreversible) — Reversible Update

$$\frac{\sigma \vdash_{expr} e \Rightarrow v \qquad v_2 = [\![\oplus]\!](v_1, v)}{\sigma \uplus \{x \mapsto v_1\} \vdash_{stmt} x \oplus= e \Rightarrow \sigma \uplus \{x \mapsto v_2\}}\ \text{AssVar}$$

- $\sigma \vdash_{expr} e \Rightarrow v$ 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 ^ .

$\Rightarrow$ It has an inverse function.

Statement $x \oplus= e$ is reversible.

Reversible Conditional

Test — Assertion

$$\frac{\sigma \vdash_{expr} e_1 \Rightarrow v_1 \ \text{is-true?}(v_1) \qquad \sigma \vdash_{stmt} s_1 \Rightarrow \sigma' \qquad \sigma' \vdash_{expr} e_2 \Rightarrow v_2 \ \text{is-true?}(v_2)}{\sigma \vdash_{stmt} \texttt{if}\ e_1\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ e_2 \Rightarrow \sigma'}\ \text{IfTrue}$$

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

$\Rightarrow$ IfTrue is reversible.

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

Reversible Loop

- Similar to reversible conditional.

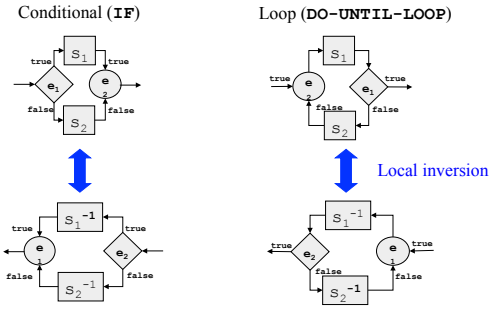
Assertion — Test

$$\frac{\sigma \vdash_{expr} e_1 \Rightarrow v_1 \ \text{is-true?}(v_1) \qquad \sigma \vdash_{loopI} (e_1, s_1, s_2, e_2) \Rightarrow \sigma' \qquad \sigma' \vdash_{expr} e_2 \Rightarrow v_2 \ \text{is-true?}(v_2)}{\sigma \vdash_{stmt} \texttt{from}\ e_1\ \texttt{do}\ s_1\ \texttt{loop}\ s_2\ \texttt{until}\ e_2 \Rightarrow \sigma'}\ \text{LoopMain}$$

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

$\Rightarrow$ LoopMain is reversible.

Local Inversion of Reversible CFOs



Local Inversion of Reversible CFOs

Conditional (**IF**)	Loop (**DO-UNTIL-LOOP**)
if e_1 then s_1 else s_2 fi e_2	from e_1 do s_1 loop s_2 until e_2
⇕	⇕ Local inversion
if e_2 then s_1^{-1} else s_2^{-1} fi e_1	from e_2 do s_1^{-1} loop s_2^{-1} until e_1

Procedure Call / Uncall

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

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

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

- Procedure call / uncall is reversible.

call id
⇕ Local inversion
uncall id

Skip and Sequence

$$\frac{}{\sigma \vdash_{stmt} \texttt{skip} \Rightarrow \sigma}\ \text{Skip}$$

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

- Skip and sequence are reversible.

s1 s2	**skip**
⇕	⇕ Local inversion
$s2^{-1}$ $s1^{-1}$	**skip**

Statement sequence is reversed.

Short Summary: Reversibility of Janus

- Janus statements are reversible.

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

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

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.

Properties of Expressions and Statements

- Equivalence Relations

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

  $s_1 \sim s_2$ 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.

Properties of Statements

- Inverse
  - Inverse statements are commutative

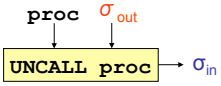
$$s_1\ s_2 \sim \epsilon \iff s_2\ s_1 \sim \epsilon$$

- Sequential inverse

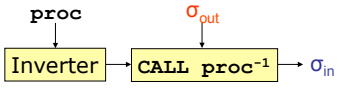
$$(s_1\ s_2)^{-1} \sim {s_2}^{-1}\ {s_1}^{-1}$$

Review: Two Approaches to Inversion of Programs

- Inverse Interpretation:



- Program Inversion:



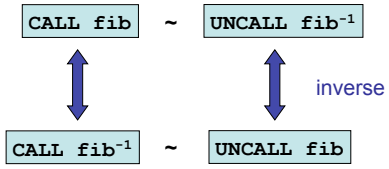
Example: Program Inversion

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

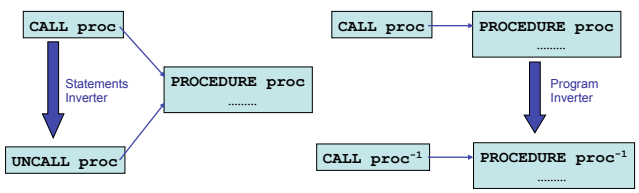
```
procedure fib
  if n=0 then x1+=1
       test  x2+=1
       else  n-=1
             call fib
             x1+=x2
             x1<=>x2
  fi x1=x2  assertion
```

Program Inversion $\mathcal{I}$

Two Ways of Invoking Procedures



Two Ways of Inversion



Statement Inverter

$$
\begin{array}{lcl}
\mathcal{I}[\![x \oplus\texttt{= } e]\!] & = & x \ \oplus'\texttt{= } e \quad \textit{where } (\oplus') = \mathcal{I}_{op}[\![\oplus]\!] \\
\mathcal{I}[\![x\texttt{[}e_l\texttt{]} \oplus\texttt{= } e]\!] & = & x\texttt{[}e_l\texttt{]} \ \oplus'\texttt{= } e \quad \textit{where } (\oplus') = \mathcal{I}_{op}[\![\oplus]\!] \\
\mathcal{I}[\![\ \texttt{if } e_1 \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi } e_2\ ]\!] & = & \texttt{if } e_2 \texttt{ then } \mathcal{I}[\![s_1]\!] \texttt{ else } \mathcal{I}[\![s_2]\!] \texttt{ fi } e_1 \\
\mathcal{I}[\![\ \texttt{from } e_1 \texttt{ do } s_1 \texttt{ loop } s_2 \texttt{ until } e_2\ ]\!] & = & \texttt{from } e_2 \texttt{ do } \mathcal{I}[\![s_1]\!] \texttt{ loop } \mathcal{I}[\![s_2]\!] \texttt{ until } e_1 \\
\mathcal{I}[\![\texttt{call } id]\!] & = & \texttt{uncall } id \\
\mathcal{I}[\![\texttt{uncall } id]\!] & = & \texttt{call } id \\
\mathcal{I}[\![\texttt{skip}]\!] & = & \texttt{skip} \\
\mathcal{I}[\![s_1\ s_2]\!] & = & \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!]
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{I}_{op}[\![\texttt{+}]\!] & = & \texttt{-} \\
\mathcal{I}_{op}[\![\texttt{-}]\!] & = & \texttt{+} \\
\mathcal{I}_{op}[\![\texttt{\^{}}]\!] & = & \texttt{\^{}}
\end{array}
$$

Statement Inverter

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

Ex.

```
n+=4
call fib
```

⟺

```
uncall fib
n-=4
```

Statement Inverter

Program Inverter

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

Properties of Program Inverter

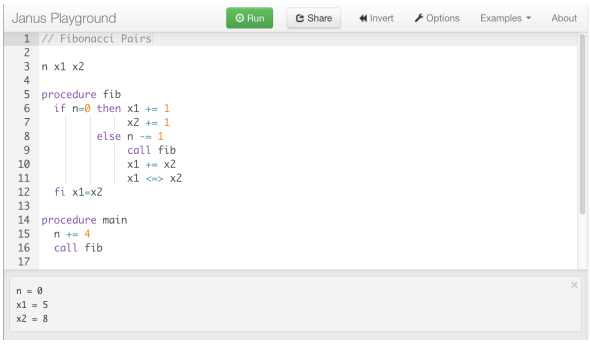
- Program inversion is performed by a recursive decent over each procedure.
- A program inverter is inverse to itself.

$$s \sim \mathcal{I}[\![\mathcal{I}[\![s]\!]]\!]$$

- The length of a program is unchanged.
- The length of computation is unchanged.
- `call fib ~ uncall fib`$^{-1}$

Janus Playground

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



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.

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).

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.