Program Inversion
and Reversible Computation
2025

Reversible Computing:
Janus (2)
A Reversible Programming Language

Robert Glück
Tetsuo Yokoyama

Today's Plan

- Programming techniques in a reversible language
- Reversible self-interpreter for Janus
  - Implementation of self-interpreter
  - Tower of self-interpreters
- An example of physical simulation in Janus
  - Discrete simulation of Schrödinger wave equation

Programming Techniques

- Each programming paradigm has its own programming techniques.
  - So do reversible languages

1. Zero-cleared copying, Zero-clearing by constant
2. Temporary stack
3. Code sharing by call and uncall
4. Call-uncall (Local Bennett's Method)

1. Zero-cleared Copying, Zero-clearing by Constant

- Zero-cleared copying:

```
{ x=0, … }
x ^= y
{ x=y, … }
```

Ex.

```
procedure main
  { n=0, … }
  n ^= 4
  { n=4, … }
  call fib
```

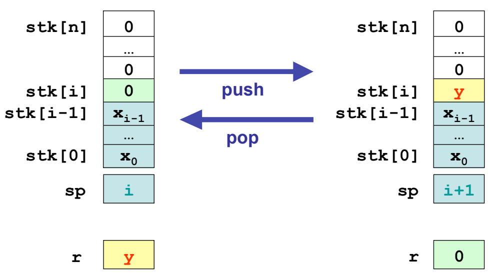
- Zero-clearing by constant:

```
{ x=y, … }
x ^= y
{ x=0, … }
```

Ex.

```
procedure main⁻¹
  uncall fib
  { n=4, … }
  n ^= 4
  { n=0, … }
```


2. Temporary Stack



2. Temporary Stack

- Array (initially zero-cleared): **tmp_stack[]**
- Stack pointer: **tmp_sp**
- Procedure (push, pop):

```
procedure alloc_tmp
  tmp_sp += 1
  tmp <=> tmp_stack[tmp_sp]
```

- Push: **call alloc_tmp** Save and clear **tmp**
- Pop: **uncall alloc_tmp** Restore cleared **tmp**

3. Code sharing by call and uncall

- The same procedure definition can be used with its inverse functionality by calling and uncalling it.

Ex.

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

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

```
procedure fib
```

Ex.

```
call alloc_tmp
```

push

```
uncall alloc_tmp
```

pop

```
procedure alloc_tmp
```


4. Call-Uncall (Local Bennett's Method)

- We need the result of a procedure **f**.
- We want to undo all other side effects the computation has had on the store.

⇒ Use the "call-uncall" program pattern.

```
call f
// copy the result of f
uncall f
```

Procedure **f** does *not* modify the variables to which the result of **f** is copied.

RTM-Interpreter in Janus (extended)

```
procedure main()
 ... RTM, tape and constants decl. and init. ...
 from  q=QS      start state
 do    call inst(q,left,s,right,q1,s1,s2,q2,pc)
       pc += 1
       if pc=PC_MAX then
          pc ^= PC_MAX                         index of next rule
       fi pc=0
 until q=QF      final state                      main loop

procedure pushtape(int s,stack stk)
 if empty(stk) && (s=BLANK) then
    s ^= BLANK   // zero-clear s
 else
    push(s,stk)
 fi empty(stk)
```



```
procedure inst(int q, stack left, int s, stack right,
               int q1, int s1, int s2, int q2, int pc)

 if (q=q1[pc]) && (s=s1[pc]) then       // Symbol rule:
    q += q2[pc]-q1[pc]                  // set q to q2[pc]
    s += s2[pc]-s1[pc]                  // set s to s2[pc]
 fi (q=q2[pc]) && (s=s2[pc])
 if (q=q1[pc]) && (s1[pc]=SLASH) then   // Move rule:
    q += q2[pc]-q1[pc]                  // set q to q2[pc]
    if s2[pc]=RIGHT then
       call   pushtape(s,left)          // push s on left
       uncall pushtape(s,right)         // pop right to s
    fi s2[pc]=RIGHT
    if s2[pc]=LEFT then
       call   pushtape(s,right)         // push s on right
       uncall pushtape(s,left)          // pop left to s
    fi s2[pc]=LEFT
 fi (q=q2[pc]) && (s1[pc]=SLASH)
```


Review: Self-Interpreter

- A self-interpreter *sint* for *L* is an *L*-interpreter written in *L*:

$$[\![sint]\!]_L [p, x] = [\![p]\!]_L \; x$$

- When *L* is a reversible language, the self-interpreter must be reversible.

Self-Interpreter for Janus

- Problem: evaluation of Janus expressions is backward nondeterministic!
- We need to implement those evaluation rules by reversible statements.

⇒ "Local Bennett's Method"

```
x += exp
```

```
call calculate_exp           Compute w/side effects
x += result_of_exp
uncall calculate_exp         Undo side effects
```

Outline of Solution

1. Simplified expressions (e.g., preprocessor)
   At most one operator in each expression.

2. Evaluation of simplified expressions.

1. Simplification of Janus Statements

```
x += (1+y)+3
```

```
t1 ^= 1
t2 ^= y
t3 ^= t1 + t2
t4 ^= 3
x  += t3 + t4
t4 ^= 3
t3 ^= t1 + t2
t2 ^= y
t1 ^= 1
```

Zero-cleared copying

Zero-clearing by constant

RHS of an assignment statement is either a
- constant,
- variable, or
- binary expression with two variables.

Encoding of Janus Programs

- Since Janus has only numerical data, we encode Janus programs in two integer arrays:
  - **type[]** Type of syntactic construct (e.g., constant)
  - **para[]** Optional parameter (e.g., value of constant)
  - **pc** Program counter
- For example, assignment `x += 5` is encoded by

type	n_{aop}^{start}	n_{aop}	n_{con}	n_{con}	n_{aop}^{end}
para	4	$n_{\mathrm{aop}}^{\mathrm{plus}}$	271	5	4

Index of variable **x** in store

State of Self-Interpreter

- Syntactic types
  n_con, n_var, n_bop_start, …
- Syntactic parameters
  n_plus, n_minus, n_xor, …
- Store of interpreted program
  sigma[]
- Temporaries
  op, arg1, arg2, tmp, …
- Temporary stack
  tmp_sp, tmp_stack[]

Skipping over Syntactic Blocks

For example, assignment `x += 5` is encoded by

type	n_{aop}^{start}	n_{aop}	n_{con}	n_{con}	n_{aop}^{end}
para	$\boxed{4}$	$n_{\mathrm{aop}}^{\mathrm{plus}}$	271	5	$\boxed{4}$

Each statement has paired offsets.

```
procedure next
  next_tmp ^= para[pc]
  pc += next_tmp
  next_tmp ^= para[pc]
  pc += 1
```

Zero-cleared copying

set **next_tmp** to 0

Zero-clearing by constant

Remark: temporary `next_tmp` needed because `pc` may not occur on both sides of an assignment.

Skipping Forward and Backward

- Skipping over syntactic blocks in both directions:

type	n_{aop}^{start}	n_{aop}	n_{con}	n_{con}	n_{aop}^{end}
para	$\boxed{4}$	$n_{\mathrm{aop}}^{\mathrm{plus}}$	271	5	$\boxed{4}$

pc

call next ↓ ↑ uncall next

type	n_{aop}^{start}	n_{aop}	n_{con}	n_{con}	n_{aop}^{end}
para	$\boxed{4}$	$n_{\mathrm{aop}}^{\mathrm{plus}}$	271	5	$\boxed{4}$

pc

Review: 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

Execution of Statements: Conditional

Test — Assertion

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

```
procedure exec_if
  pc += 1
  call eval
  if result
  then uncall eval
       pc -= 1
       call next  // to then
       call exec
       call next  // to fi
       pc += 1
       call eval
  fi result
  uncall eval
  pc -= 1
  call next  // to end of stmt
  pc -= 1
```

Evaluation of the test

call-uncall

Undoing the side effects of evaluating the test

```
else uncall eval
     pc -= 1
     call next  // to then
     call next  // to else
     call exec
     pc += 1
     call eval
```

Evaluation of the assertion

call-uncall

Undoing the side effects of evaluating the assertion

Execution of Statements: Loop

- Similar to conditional

Execution of Statements: Procedure Call

Execution of body of procedure *id*

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

```
procedure exec_call
  call exec_call_pc_swap
  call exec
  uncall next
  uncall exec_call_pc_swap
```

Save **pc** and set new **pc** to body of procedure.

call-uncall

Restore **pc**

```
procedure exec_call_pc_swap
  tmp ^= para[pc]
  pc <=> tmp
  call alloc_tmp
```

pc:=para[pc]

temporary stack

Execution of Statements: Procedure Uncall

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

$$\frac{\sigma' \vdash_{stmt} \Gamma(id) \Rightarrow \sigma}{\sigma \vdash_{stmt} \texttt{uncall}\ id \Rightarrow \sigma'}\ \text{Uncall}$$

- How to implement call and uncall in the interpreter?

```
procedure exec_uncall
          ?
```

Code-sharing by call-uncall

Review: Expression Evaluation is Irreversible

Judgment: $\sigma \vdash_{expr} e \Rightarrow v$ (Store, Exp, Val)

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

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

where $\odot \in \{\ \texttt{*}, \texttt{/}, \texttt{\&}, <, >, \dots\ \}$
are non-injective operators

Reversible Evaluation of Expressions

- Case: constants

Reversible update — Evaluation of RHS (irreversible)

$$\frac{v' = [\![\oplus]\!](v, \boxed{[\![c]\!]})}{\sigma \uplus \{x \mapsto v\} \vdash_{stmt} x\ \boxed{\oplus=}\ \boxed{c} \Rightarrow \sigma \uplus \{x \mapsto v'\}}$$

- Case: variables

$$\frac{v' = [\![\oplus]\!](v, \boxed{\sigma(y)})}{\sigma \uplus \{x \mapsto v\} \vdash_{stmt} x\ \boxed{\oplus=}\ \boxed{y} \Rightarrow \sigma \uplus \{x \mapsto v'\}}$$

- Case: binary operators

$$\frac{v' = [\![\oplus]\!](v, \boxed{[\![\odot]\!](\sigma(y_1), \sigma(y_2))})}{\sigma \uplus \{x \mapsto v\} \vdash_{stmt} x\ \boxed{\oplus=}\ \boxed{y_1 \odot y_2} \Rightarrow \sigma \uplus \{x \mapsto v'\}}$$

Note: simplified expressions have at most one operator

Evaluation of Expressions: Dispatch

```
procedure eval
  if type[pc] = n_con
  then result ^= para[pc]
  else if type[pc] = n_var
       then result ^= sigma[para[pc]]
       else if type[pc] = n_bop_start
            then call eval_bop
                 call next
                 pc-=1
            else error
            fi type[pc] = n_bop_end
       fi type[pc] = n_var
  fi type[pc] = n_con
  pc+=1
```

Constant: value in **para[]**

Variable: look-up in **sigma[]**

Binary operator: call sub-procedure

Dispatch depends on **type[pc]**

Assertions on **type[pc]**

Evaluation of Expressions: Binary Operators

```
procedure eval_bop
  call eval_bop_args
  if op = n_plus
  then tmp ^= arg1 + arg2
  else if op = n_minus
       then tmp ^= arg1 - arg2
       else if op = n_xor
            then tmp ^= arg1 ^ arg2
            // ...
            fi op = n_xor
       fi op = n_minus
  fi op = n_plus
  uncall eval_bop_args
  result <=> tmp
```

Evaluate both arguments and return results in **arg1** and **arg2**

$$v' = \boxed{[\![\odot]\!]}(\underbrace{\sigma(y_1)}_{\text{arg1}}, \underbrace{\sigma(y_2)}_{\text{arg2}})$$

Side-effects are undone

call-uncall

The underlying operation is evaluated.

Evaluation of Expressions: Arguments

```
procedure eval_bop_args
  pc += 1
  op ^= para[pc]
  pc += 1
  call eval
  arg1 <=> result
  call eval
  arg2 <=> result
```

Determine the binary operator

Evaluate 1st argument

Evaluate 2nd argument

Execution of Statements: Reversible Update

$$\frac{v' = f(\sigma, \oplus, v, e)}{\sigma \uplus \{x \mapsto v\} \vdash_{stmt} x\ \boxed{\oplus=}\ \boxed{e} \Rightarrow \sigma \uplus \{x \mapsto v'\}}$$

```
procedure exec_aop
  call exec_aop_args
  call exec_aop_upd
  uncall exec_aop_args
  call next    // to end of stmt
  pc -= 1
```

Evaluation of RHS

Reversible Update

call-uncall

Execution of Statements: Dispatch

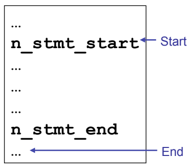
```
procedure exec1
  if type[pc] = n_aop_start
  then call exec_aop
  else if type[pc] = n_if_start
       then call exec_if
       else if type[pc] = n_from_start
            then call exec_from
            else if type[pc] = n_call
                 then call exec_call
                 else if type[pc] = n_uncall
                      then call exec_uncall
                      else if type[pc] = n_skip
                           else error
                           fi type[pc] = n_skip
                      fi type[pc] = n_uncall
                 fi type[pc] = n_call
            fi type[pc] = n_until_end
       fi type[pc] = n_fi_end
  fi type[pc] = n_aop_end
```

Dispatched depending on **type[pc]**

Execution of Statements: Top Level

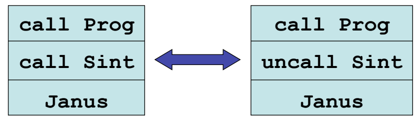
- Start from `n_stmt_start` and end with `n_stmt_end`
- In each iteration, a statement is executed by `exec1`

```
procedure exec
    from  type[pc] = n_stmt_start
    do    pc += 1
    loop  call exec1
    until type[pc] = n_stmt_end
    pc += 1
```



Short Summary: Self-Interpreter

- Implementation completed
- History-free
  - Uses only local Bennett's method
- Constant space
- Reversible self-interpreter



Experiments with Janus

- Many physical models describe reversible processes.
- Schrödinger Wave Equation
  - The fundamental equation of physics for describing quantum mechanical behavior.
  - F $\mathcal{X}_{i,n+1} = \mathcal{X}_{i,n} + \alpha_i \mathcal{Y}_{i,n} - \epsilon(\mathcal{Y}_{i+1,n} + \mathcal{Y}_{i-1,n})$
    $\mathcal{Y}_{i,n+1} = \mathcal{Y}_{i,n} - \alpha_i \mathcal{X}_{i,n+1} + \epsilon(\mathcal{X}_{i+1,n+1} + \mathcal{X}_{i-1,n+1})$
  - C $\mathcal{X}_{i,n} = \mathcal{X}_{i+128,n}$, $\mathcal{Y}_{i,n} = \mathcal{Y}_{i+128,n}$

† E. Fredkin. Feynman, Barton and the reversible Schrödinger difference equation. Feynman and computation: exploring the limits of computers, pages 337-348,1999.

M. P. Frank. Reversibility for Efficient Computing. PhD thesis, EECS Dept., MIT, 1999.

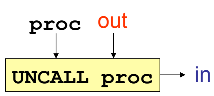
Simulation Program: `Sch`

$$\mathcal{X}_{i,n+1} = \mathcal{X}_{i,n} + \alpha_i \mathcal{Y}_{i,n} - \epsilon(\mathcal{Y}_{i+1,n} + \mathcal{Y}_{i-1,n})$$

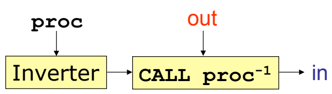
```
procedure main
  ... // initialize arrays
  from  n=0
  loop  call step
        n += 1
  until n=maxn

procedure step
  call stepX
  call stepY

procedure updateX
  X[i] += alpha[i] */ Y[i]
  X[i] -= epsilon */
          (Y[(i+1)%128] +
           Y[(i-1)%128])

procedure stepX
  from  i=0
  loop  call updateX
        i += 1
  until i=128
  i -= 128
```


Review: Two Approaches to Inversion

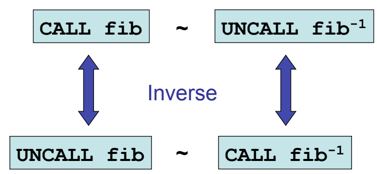
- Inverse interpretation of a procedure (one stage):

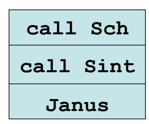


- Program inversion of a procedure (two stages):



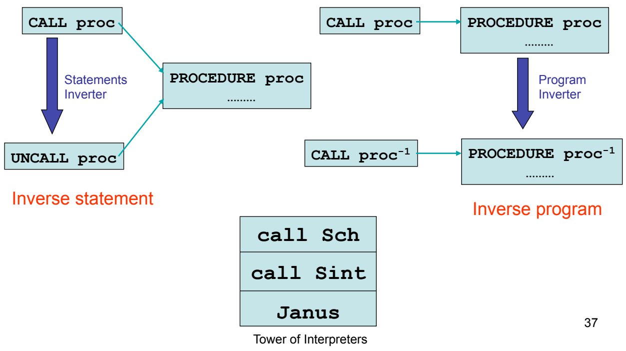
Review: Two ways of Invoking Procedures





Tower of Interpreters

Review: Two Ways of Inversion



Inverse Call and Program Inversion

- Forward (2^2 / 2 = 2 possibilities)

call Sch	~	uncall Sch^{-1}
Janus		Janus

- Backward (2^2 / 2 = 2 possibilities)

call Sch^{-1}	~	uncall Sch
Janus		Janus

Inverse Call and Program Inversion

- Forward (2^4 / 2 = 8 possibilities)

call Sch		call Sch^{-1}		uncall Sch^{-1}		uncall Sch
call Sint	~	call Sint^{-1}	~	call Sint	~	call Sint^{-1}
Janus		Janus		Janus		Janus
≀		≀		≀		≀
call Sch^{-1}		call Sch		call Sch^{-1}		uncall Sch^{-1}
uncall Sint	~	uncall Sint^{-1}	~	uncall Sint	~	uncall Sint^{-1}
Janus		Janus		Janus		Janus

Inverse Call and Program Inversion

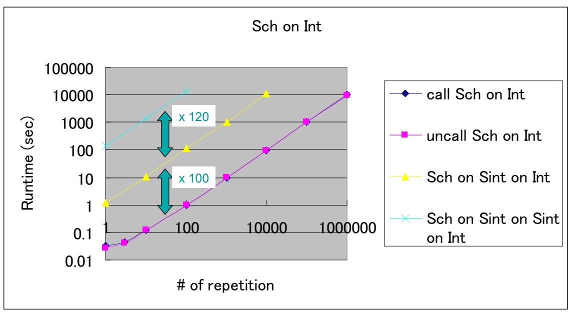
- Backward (2^4 / 2 = 8 possibilities)

uncall Sch		uncall Sch^{-1}		call Sch^{-1}		call Sch
call Sint	~	call Sint^{-1}	~	call Sint	~	call Sint^{-1}
Janus		Janus		Janus		Janus
≀		≀		≀		≀
call Sch		call Sch^{-1}		call Sch^{-1}		uncall Sch
uncall Sint	~	uncall Sint^{-1}	~	uncall Sint^{-1}	~	uncall Sint^{-1}
Janus		Janus		Janus		Janus

Tower of Interpreters

		call Sch	
	call Sch	call Sint	
call Sch	call Sint	call Sint	...
Janus	Janus	Janus	

x 100 ~ 160
Interpretive overhead



Summary

- To implement the Janus self-interpreter, the operational semantics rules for expressions need to be implemented reversibly.
- Reversible programming paradigm has its own programming techniques.
- Janus self-interpreter realizes non-standard interpreter hierarchy.

Exercises: Free-Falling Object