Wrongforth - A reversible Forth

Gerald Wodni
gerald.wodni@gee.at
TU Wien

Euroforth 2009

Motivation

- Physical limits of Computation
- Robert Glück's course - Program Inversion and Reversible Computation [1][2]
- Porting the idea of Janus (A reversible language) to another language
- Forth is close to machine level, reversible hardware?

Landauer Limit* [1][5]

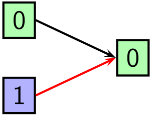
- Every logical state corresponds to a physical state of a device
- Physical information cannot be destroyed, only transformed into heat
- Loss of information by 1 bit → releases 3 zeptojoules
- 2003 ITRS Perspective: Practical limit for CMOS reached in 2030?
- Arbitraty low energy consumption

---

*Rolf Landauer 1961

Preserve Information

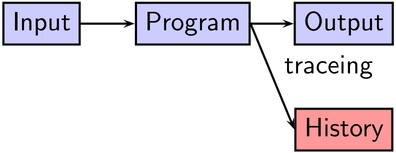
- $2 + 3 = 5$
- $5 = x + y$: $1 + 4$, $4 + 1$, $2 + 3$, $3 + 2$
- Saving an operand makes the operation reversible
- $2 + 3 = 5(2) \rightarrow 5 - 2 = 3$



clear bit

Types of Inversion

- Tracing*


Input

Program

Output

traceing

History

  - History grows propotionally to length of computation
  - Forward programming as usual
  - Backtracking, input can only be restored with history

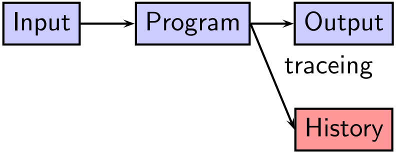
---

[*] Bennet:73

[†] Dijkstra:78, Gries:81

Types of Inversion

- Tracing[*]

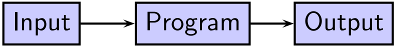


Input → Program → Output

traceing

History

  - History grows propotionally to length of computation
  - Forward programming as usual
  - Backtracking, input can only be restored with history
- Stand-Alone[†]



Input → Program → Output

  - No History → also applicable in low memory machines
  - For-/Backward programming
  - Algorithm is inverted:
    input can be generated from any output (e.g. writing zip and implicitly getting unzip)

---

[*] Bennet:73

[†] Dijkstra:78, Gries:81

Reversible Hardware*[2]

- Direction Bit: changes the incrementation direction of the PC
- Paired Branches:
  1: BRA 2
  2: NOP
  3: BRA -2

---

*Holger Bock Axelsen, Robert Glück 2007

RVM-FORTH* [3]

- Guards & Choices
- Reversible primitives (e.g. !_ C!_ +!_)
- Backtracking

---

*Bill Stoddart 2004

Janus* [1]

- No destructive assignments (:=)
- Only operators with a reversible counterpart (+ -, * /)
- Each condition has a matching assertion
- Call/Uncall

---

*Tetsuo Yokoyama, Robert Glück 2007

Fibonacci [1]

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

Fibonacci [1]

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

Non-destructive updates

Fibonacci [1]

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

Swap operator

Fibonacci [1]

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

Condition-assertion pair

Fibonacci [1]

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

	n	x1	x2
●	0	0	0
	3	0	0
	2	0	0
	1	0	0
	0	0	0
	0	1	1
	0	2	1
	0	1	2
	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
●	3	0	0
	2	0	0
	1	0	0
	0	0	0
	0	1	1
	0	2	1
	0	1	2
	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
●	2	0	0
	1	0	0
	0	0	0
	0	1	1
	0	2	1
	0	1	2
	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
●	1	0	0
	0	0	0
	0	1	1
	0	2	1
	0	1	2
	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
	1	0	0
●	0	0	0
	0	1	1
	0	2	1
	0	1	2
	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
	1	0	0
	0	0	0
●	0	1	1
	0	2	1
	0	1	2
	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
	1	0	0
	0	0	0
	0	1	1
●	0	2	1
	0	1	2
	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
	1	0	0
	0	0	0
	0	1	1
	0	2	1
●	0	1	2
	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
	1	0	0
	0	0	0
	0	1	1
	0	2	1
	0	1	2
●	0	3	2
	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
	1	0	0
	0	0	0
	0	1	1
	0	2	1
	0	1	2
	0	3	2
●	0	2	3
	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
	1	0	0
	0	0	0
	0	1	1
	0	2	1
	0	1	2
	0	3	2
	0	2	3
●	0	5	3
	0	3	5

Fibonacci [1]

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

	n	x1	x2
	0	0	0
	3	0	0
	2	0	0
	1	0	0
	0	0	0
	0	1	1
	0	2	1
	0	1	2
	0	3	2
	0	2	3
	0	5	3
●	0	3	5

Language Design

- $+$ not reversible (loses an operand)
- over not reversible (equals drop in reverse computation)
- over$+$ reversible
- literals not reversible
- stack stays the same (use variables ?)
- postfix: operands are after the operator in rerverse computation

Forth Implementation

- Dual Compilation
  - Primitives are compiled with for- & backward semantics
  - Twice the size in memory
- Reversible Computation
  - Primitives switch their semantics based on execution direction
  - Small memory footprint (forward + assertions)
  - No information loss

Fibonacci

- Dual Compilation

```
\--- forward ---
1 direction !
:  fib ( -- ) recursive
   n @ 0= _if
     x1 1 +=
     x2 1 +=
   _else
     n 1 -=
     fib
     x1 x2 @ +=
     x1 x2 <=>
   x1 @ x2 @ = _then
;
```

```
\--- backward ---
0 direction !
:  fib-1 ( -- ) recursive
   x1 @ x2 @ = _then
     x1 x2 <=>
     x1 x2 @ +=
     fib-1
     n 1 -=
   _else
    x2 1 +=
    x1 1 +=
   n @ 0= _if
;
```

Fibonacci - Reversible Computation

```
:  callfib fibptr [resume] ;

:  fib ( n x1 x2 -- n x1 x2 )
   beg
     bwdcheck third0= _if
       _1+
       _swap
       _1+
       _swap
     _else
       _rot
       _1-
       _-rot
       callfib
       _swap
       _over+
     _then 2dup= fwdcheck
   end ;
```

Operators

- Dual Compilation

```
:  _+ iffwd POSTPONE + else POSTPONE - then ; immediate
:  _- iffwd POSTPONE - else POSTPONE + then ; immediate
```

- Reversible Computation

```
:  _1+ forewards?  if 1+ else 1- then [resume] ;
:  _1- forewards?  if 1- else 1+ then [resume] ;
```

Control Structures

- Dual Compilation

```
:   _if iffwd POSTPONE if
    else POSTPONE drop POSTPONE then
    then ; immediate

:   _then iffwd POSTPONE drop POSTPONE then
    else POSTPONE invert POSTPONE if
    then ; immediate

:   _else POSTPONE else ; immediate
```

Control Structures

- Dual Compilation

```
:  _if iffwd POSTPONE if
   else POSTPONE drop POSTPONE then
   then ; immediate

:  _then iffwd POSTPONE drop POSTPONE then
   else POSTPONE invert POSTPONE if
   then ; immediate

:  _else POSTPONE else ; immediate
```

Control Structures

- Dual Compilation

```
:   _if iffwd POSTPONE if
    else POSTPONE drop POSTPONE then
    then ; immediate

:   _then iffwd POSTPONE drop POSTPONE then
    else POSTPONE invert POSTPONE if
    then ; immediate

:   _else POSTPONE else ; immediate
```

Control Structures

- Reversible Computation

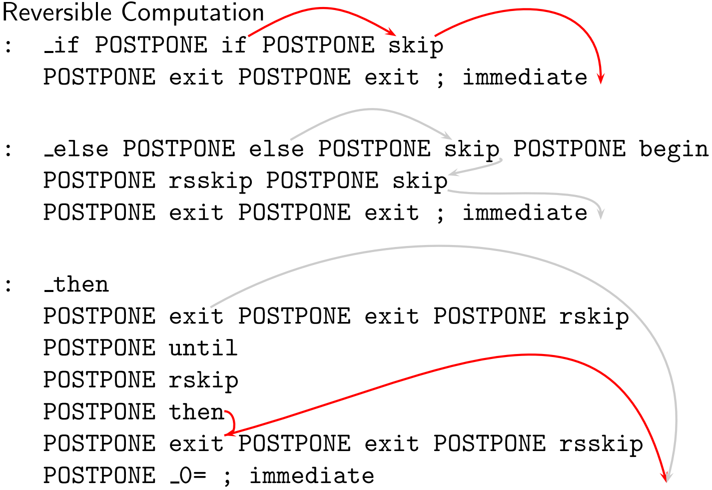
```
:  _if POSTPONE if POSTPONE skip
   POSTPONE exit POSTPONE exit ; immediate

:  _else POSTPONE else POSTPONE skip POSTPONE begin
   POSTPONE rsskip POSTPONE skip
   POSTPONE exit POSTPONE exit ; immediate

:  _then
   POSTPONE exit POSTPONE exit POSTPONE rskip
   POSTPONE until
   POSTPONE rskip
   POSTPONE then
   POSTPONE exit POSTPONE exit POSTPONE rsskip
   POSTPONE _0= ; immediate
```

Control Structures

- Reversible Computation



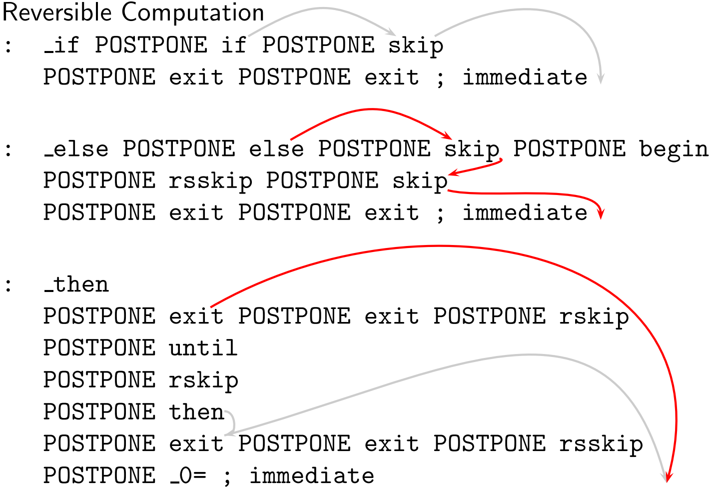
```
:   _if POSTPONE if POSTPONE skip
    POSTPONE exit POSTPONE exit ; immediate

:   _else POSTPONE else POSTPONE skip POSTPONE begin
    POSTPONE rsskip POSTPONE skip
    POSTPONE exit POSTPONE exit ; immediate

:   _then
    POSTPONE exit POSTPONE exit POSTPONE rskip
    POSTPONE until
    POSTPONE rskip
    POSTPONE then
    POSTPONE exit POSTPONE exit POSTPONE rsskip
    POSTPONE _0= ; immediate
```

Control Structures

- Reversible Computation



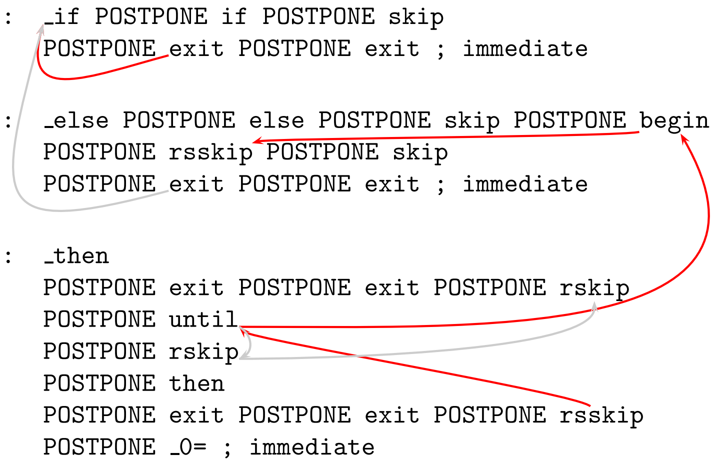
```
:   _if POSTPONE if POSTPONE skip
    POSTPONE exit POSTPONE exit ; immediate

:   _else POSTPONE else POSTPONE skip POSTPONE begin
    POSTPONE rsskip POSTPONE skip
    POSTPONE exit POSTPONE exit ; immediate

:   _then
    POSTPONE exit POSTPONE exit POSTPONE rskip
    POSTPONE until
    POSTPONE rskip
    POSTPONE then
    POSTPONE exit POSTPONE exit POSTPONE rsskip
    POSTPONE _0= ; immediate
```

Control Structures

- Reversible Computation



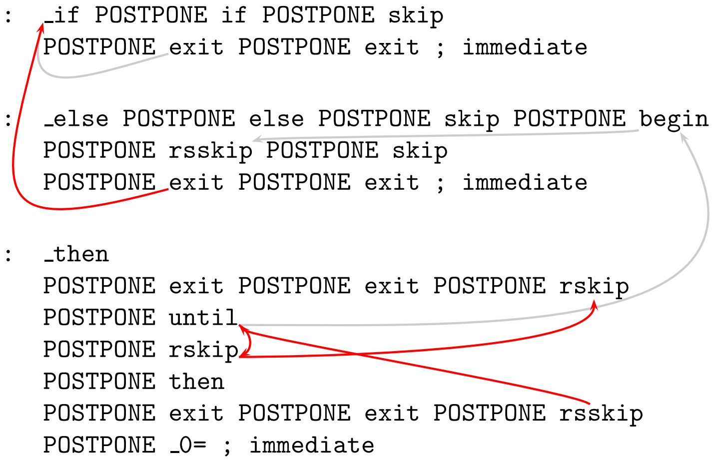
```
:   _if POSTPONE if POSTPONE skip
    POSTPONE exit POSTPONE exit ; immediate

:   _else POSTPONE else POSTPONE skip POSTPONE begin
    POSTPONE rsskip POSTPONE skip
    POSTPONE exit POSTPONE exit ; immediate

:   _then
    POSTPONE exit POSTPONE exit POSTPONE rskip
    POSTPONE until
    POSTPONE rskip
    POSTPONE then
    POSTPONE exit POSTPONE exit POSTPONE rsskip
    POSTPONE _0= ; immediate
```

Control Structures

- Reversible Computation



```
:   _if POSTPONE if POSTPONE skip
    POSTPONE exit POSTPONE exit ; immediate

:   _else POSTPONE else POSTPONE skip POSTPONE begin
    POSTPONE rsskip POSTPONE skip
    POSTPONE exit POSTPONE exit ; immediate

:   _then
    POSTPONE exit POSTPONE exit POSTPONE rskip
    POSTPONE until
    POSTPONE rskip
    POSTPONE then
    POSTPONE exit POSTPONE exit POSTPONE rsskip
    POSTPONE _0= ; immediate
```

Conclusion

- Dual Compilation
    - As fast as common Forth
    - Easy to implement
    - Works in ANS-Forth
- Reversible Computation
    - Suitable for reversible hardware [2]
    - Literals backwards sematics
    - Manipulates the return-stack
    - Would benefit from changes in the Forth-VM (Direction Register, Paired Branches)
- Both
    - Could save implementation time: Compression, FFT, ... only need to be implemented once

References

1 Tetsuo Yokoyama, Robert Glück. A Reversible Programming Language and its Invertible Self-Interpreter. 2007.

2 Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. Reversible Machine Code and Its Abstract Processor Architecture. 2007.

3 Bill Stoddart. RVM-FORTH, a Reversible Virtual Machine. 2004.

4 Neal Crook, Anton Ertl, David Kuehling, Bernd Paysan, Jens Wilke. GForth-Manual. 1995-2008.

5 Rolf Landauer. Irreversibility and heat generation in the computing process. 1961.