Reversible Programming Languages

Vikraman Choudhury

April, 2018

1 Reversible Programming Languages

Moore's Law observes that the number of electronic components in a hardware circuit doubles approximately every two years. This iteration has been going on for decades and has resulted in a marked increase in power consumption of hardware circuits. With classical models of computation, there exists a theoretical limit on the energy consumption of any computation, the *von Neumann-Landauer* limit. Landauer's principle states that,

> ...any logically irreversible manipulation of information, such as the erasure of a bit or the merging of two computation paths, must be accompanied by a corresponding entropy increase in non-information-bearing degrees of freedom of the information-processing apparatus or its environment Landauer [1961]

An erasure of information in a closed system is always accompanied by an increase in energy consumption. To erase n bits of information $n \cdot k_B \cdot ln\, 2$ amount of energy is required, where T is the temperature of the circuit in Kelvin, and k_B is the *Boltzmann constant.* This motivates the need to develop *reversible* models of computation, where information or bits are never erased, so that reversible computing systems are not subject to the von Neumann-Landauer limit.

To realize a fully reversible computation system, we need reversibility at every level of abstraction. Much progress has been made on achieving reversibility at the circuit and gate levels, but not at the high-level programming language level. A reversible high-level language should be able to be compiled down to a low-level reversible assembly language without significant overhead.

This survey explores some reversible programming languages and their semantics.

Janus Grammar

$$
\begin{array}{rcll}
prog & ::= & p_{main}\ p^{*} & \text{(program)} \\
t & ::= & \mathbf{int} \mid \mathbf{stack} & \text{(data type)} \\
p_{main} & ::= & \mathbf{procedure\ main}\ \mathbf{()}\ (\mathbf{int}\ x(\mathbf{[}\overline{n}\mathbf{]})^{?} \mid \mathbf{stack}\ x)^{*}\ s & \text{(main procedure)} \\
p & ::= & \mathbf{procedure}\ q\,\mathbf{(}t\ x,\ \ldots,\ t\ x\mathbf{)}\ s & \text{(procedure definition)} \\
s & ::= & x\ \odot\mathbf{=}\ e \mid x\mathbf{[}e\mathbf{]}\ \odot\mathbf{=}\ e & \text{(assignment)} \\
 & \mid & \mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ s\ \mathbf{fi}\ e & \text{(conditional)} \\
 & \mid & \mathbf{from}\ e\ \mathbf{do}\ s\ \mathbf{loop}\ s\ \mathbf{until}\ e & \text{(loop)} \\
 & \mid & \mathbf{push}\,\mathbf{(}x,\ x\mathbf{)} \mid \mathbf{pop}\,\mathbf{(}x,\ x\mathbf{)} & \text{(stack modification)} \\
 & \mid & \mathbf{local}\ t\ x\ \mathbf{=}\ e\quad s\quad \mathbf{delocal}\ t\ x\ \mathbf{=}\ e & \text{(local variable block)} \\
 & \mid & \mathbf{call}\ q\,\mathbf{(}x,\ \ldots,\ x\mathbf{)} \mid \mathbf{uncall}\ q\,\mathbf{(}x,\ \ldots,\ x\mathbf{)} & \text{(procedure invocation)} \\
 & \mid & \mathbf{skip} \mid s\ s & \text{(statement sequence)} \\
e & ::= & \overline{n} \mid x \mid x\mathbf{[}e\mathbf{]} \mid e\ \otimes\ e \mid \mathbf{empty}\,\mathbf{(}x\mathbf{)} \mid \mathbf{top}\,\mathbf{(}x\mathbf{)} \mid \mathbf{nil} & \text{(expression)} \\
\odot & ::= & \mathbf{+} \mid \mathbf{-} \mid \mathbf{\hat{}} & \text{(operator)} \\
\otimes & ::= & \odot \mid \mathbf{*} \mid \mathbf{/} \mid \mathbf{\%} \mid \mathbf{\&} \mid \mathbf{|} \mid \mathbf{\&\&} \mid \mathbf{||} \mid \mathbf{<} \mid \mathbf{>} \mid \mathbf{=} \mid \mathbf{!=} \mid \mathbf{<=} \mid \mathbf{>=} & \text{(operator)}
\end{array}
$$

Figure 1: EBNF grammar for Janus

1.1 Janus

The reversible programming language *Janus* (named after the two-faced Greco-Roman god of beginnings and endings) was created by Cristopher Lutz and Howard Derby at Caltech in 1982 Lutz [1986]. It was later rediscovered and formalized in Yokoyama and Glück [2007] and some modifications were suggested in Yokoyama et al. [2008]. We consider the modified version of the language here.

Janus is a procedural language where every program statement is locally-invertible. The data types are restricted to integers, integer arrays of fixed size, and integer stacks of dynamic size. Procedures are made of program statements of different kinds.

For a conditional statement, there is a branch condition and an exit assertion expression. The exit assertion is used to reversibly join the two paths of computation in the branches. Similarly, for a loop statement, there is an entry assertion

and an exit condition. When executed in reverse, the exit condition serves as the entry assertion and vice versa. Stack updates are done using *push* and *pop* which are forced to be inversions of each other, by presupposing that the variable being pushed to is zero-cleared, which happens on popping.

For a variable update to be reversible, the original store should remain reachable by subsequent uncomputation. So the only updates that are allowed are the ones which are injective in their first argument and have a defined inverse. The expression being updated with cannot depend in any way on the variable being updated. This is enforced by restrictions on binding and scoping. A variable identifier on the left hand side cannot occur on the right hand side, and no two identifiers can point to the same location in memory. Finally, *call* and *uncall* are used to invoke procedures in the forwards and backwards direction.

Computationally, Janus is known to be r-Turing complete in Yokoyama et al. [2008], because it can simulate any reversible Turing machines.

1.2 R

The reversible programming language R (not the statistical programming language of the same name) is an imperative reversible language developed at MIT in 1997 Frank [1997]. It is a compiled language, with an S-expression syntax, which targets the Pendulum reversible instruction set 1.3.

The data types of R are restricted to integers and integer arrays, and the syntax is similar to Janus, allowing forward and backward execution of procedures.

The *if* statement requires that the value of the conditional expression is the same before and after the conditional is executed, which allows reversing the direction of execution so that the right branch can be chosen. It should be noted that this is equivalent to conditionals in Janus with the conditional expression being used as the entry and exit assertion. The same holds for the *for* loop, where the initial and terminal values of the iteration variable should be the same before and after the loop.

The *let* statement is a local assignment which is guaranteed to be reversible by ensuring that the variable is zero-cleared when it goes out of scope. This is equivalent to the local/delocal statements in Janus. Modifications to variables are restricted to increment, negate, swap, and update statements, which works similarly as in Janus.

R Grammar

prog	::=	s^*	(program)
s	::=	**(defmain** *progname* s^***)**	(main routine)
	$\mid$	**(defsub** *subname* **(**$name^*$**)** s^***)**	(subroutine)
	$\mid$	**(defword** *name* $\overline{n}$**)**	(global variable)
	$\mid$	**(defarray** *name* $\overline{n}^*$**)**	(global array)
	$\mid$	**(call** *subname* e^***)**	(call subroutine)
	$\mid$	**(rcall** *subname* e^***)**	(reverse-call subroutine)
	$\mid$	**(if** *e* **then** s^***)**	(conditional)
	$\mid$	**(for** *name* **=** *e* **to** *e* s^***)**	(loop)
	$\mid$	**(let (***name* **<-** *e***)** s^***)**	(variable binding)
	$\mid$	**(printword** *e***)** $\mid$ **(println)**	(output)
	$\mid$	**(***loc* **++)** $\mid$ **(-** *loc***)**	(increment/negate)
	$\mid$	**(***loc* **<->** *loc***)** $\mid$ **(***loc* $\odot$ *e***)**	(swap/update)
loc	::=	*name* $\mid$ **(*** *e***)** $\mid$ **(***e* **_** *e***)**	(location)
e	::=	*loc* $\mid$ **(***e* $\otimes$ *e***)** $\mid$ $\overline{n}$	(expression)
$\odot$	::=	**+=** $\mid$ **-=** $\mid$ **^=** $\mid$ **<=<** $\mid$ **>=>**	(update operator)
$\otimes$	::=	**+** $\mid$ **-** $\mid$ **&** $\mid$ **<<** $\mid$ **>>** $\mid$ ***/**	(expression operator)
	$\mid$	**=** $\mid$ **<** $\mid$ **>** $\mid$ **<=** $\mid$ **>=** $\mid$ **!=**	(relational operator)

Figure 2: EBNF grammar for R

1.3 PISA

The Pendulum microprocessor and the Pendulum ISA (PISA) is a logically reversible computer architecture created at MIT by Vieri [1995]. It resembles a mix of PDP-8 and RISC and is the first reversible programmable processor architecture.

i	i^{-1}
ADD r_1 r_2	**SUB** r_1 r_2
SUB r_1 r_2	**ADD** r_1 r_2
ADDI r c	**ADDI** r $-c$
RL r c	**RR** r c
RR r c	**RL** r c
RLV r_1 r_2	**RRV** r_1 r_2
RRV r_1 r_2	**RLV** r_1 r_2

Figure 3: Inversion rules for PISA instructions, all other instructions are self-inverse

The interesting bit is the existence of a *direction bit (DIR)* special-purpose register, alongwith the *program counter (PC)* and the *branch register (BR)* for control flow. DIR is either +1 or -1 depending on the direction of execution. If BR is 0, DIR gets added to PC. If BR is non-zero, the product of BR and DIR is added instead.

The uncoditional jump instructions *BRA* and *RBRA* (reverse BRA) work as usual but also modify the DIR bit to implement forward or reverse call functionality. The corresponding addition/subtraction/bit operations are reversed according to the value of the DIR bit.

SWAPBR allows direct control of the BR register to implement dynamic jumps. *EXCH* and *DATA* can be used to manipulate memory cells directly. The bit wise operations work as usual, but the result is stored in a third register to allow reversibility.

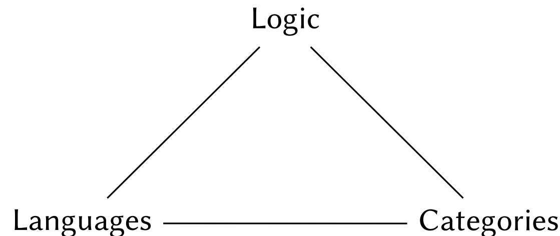
A formalization of the control flow logic is presented in Axelsen et al. [2007]. A translation from Janus to PISA is presented in Axelsen [2011] and a translation from R to PISA in Frank [1997].

1.4 Remarks

Our survey of reversible high level languages shows that they are restricted to finite integer and array datatypes, and first order procedures. This calls for more expressive reversible higher-order languages with a menagerie of inductive or recursive datatypes. A purely functional reversible language should allow a more principled reasoning about effects, for example, using monads as are prevalent in purely functional languages Moggi [1991].

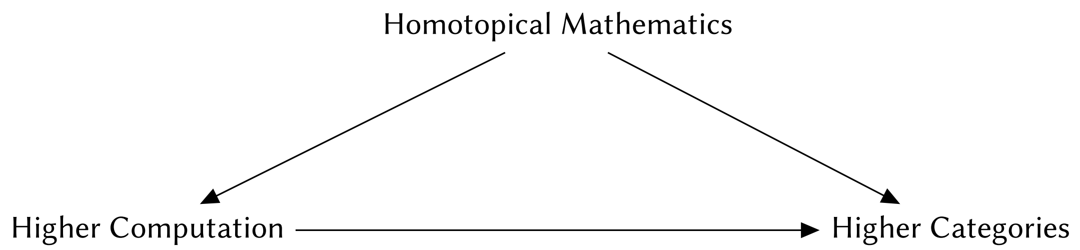
Finite types are first class citizens in the *Pi* family of reversible languages in Bowman et al. [2011], and recursive datatypes in Theseus James and Sabry [2014], which also embeds a first order fragment of the simply typed lambda calculus James and Sabry [2012]. It is conjectured in Carette and Sabry [2016] that these languages are complete with respect to weak equivalences on finite sets. Frobenius monads are known to give semantics to effectful reversible computations Heunen and Karvonen [2015], but they have not been explored in the design of a reversible programming language.

Computational trinitarianism nLab is a central principle of programming language design, as preached by Robert Harper. *Logic*, *Languages*, and *Categories* are three manifestations of one central notion of computation. Once instance of this is the duality between mathematics, computation and categories, as evident in the case of type theory.

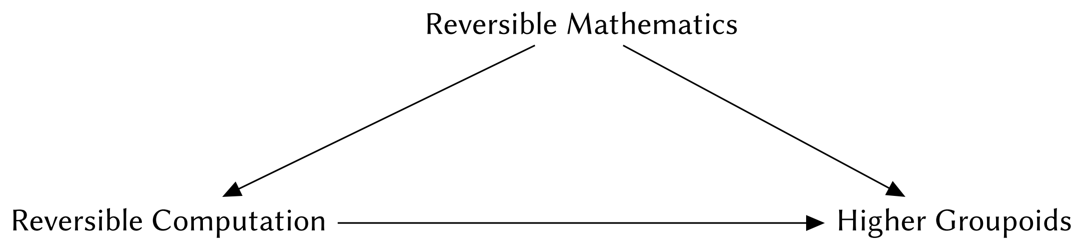


As suggested by Michael Shulman, this dogma generalises to higher dimensional type theory, using the dualities between homotopical mathematics, higher computation, and higher categories, which are well studied in Homotopy Type Theory.

One can conjecture if such a principle can be established for reversible computation. There is a duality between the syntax of a reversible programming language (as in Pi/Theseus) and higher groupoids Carette et al. [2017]. It remains to be seen whether this triangle can be completed by finding analogues



in mathematics which rely on reversible principles, such as isomorphisms or equivalences.



References

Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM journal of research and development*, 5(3):183–191, 1961.

Christopher Lutz. Janus: a time-reversible language. *Letter to R. Landauer.*, 1986.

Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 144–153. ACM, 2007.

Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In *Proceedings of the 5th conference on Computing frontiers*, pages 43–54. ACM, 2008.

Michael P Frank. The r programming language and compiler. Technical report, MIT Reversible Computing Project Memo, 1997.

Carlin James Vieri. *Pendulum–a reversible computer architecture.* PhD thesis, Massachusetts Institute of Technology, 1995.

Holger Bock Axelsen, Robert Glück, and Tetsuo Yokoyama. Reversible machine code and its abstract processor architecture. In *International Computer Science Symposium in Russia*, pages 56–69. Springer, 2007.

Holger Bock Axelsen. Clean translation of an imperative reversible programming language. In *International Conference on Compiler Construction*, pages 144–163. Springer, 2011.

Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

William J Bowman, Roshan P James, and Amr Sabry. Dagger traced symmetric monoidal categories and reversible programming. 2011.

RP James and A Sabry. Theseus: a high level language for reversible computing, work-in-progress report at rc (2014), 2014.

Roshan P James and Amr Sabry. Isomorphic interpreters from logically reversible abstract machines. In *International Workshop on Reversible Computation*, pages 57–71. Springer, 2012.

Jacques Carette and Amr Sabry. Computing with semirings and weak rig groupoids. In *European Symposium on Programming Languages and Systems*, pages 123–148. Springer, 2016.

Chris Heunen and Martti Karvonen. Reversible monadic computing. *Electron. Notes Theor. Comput. Sci.*, 319(C):217–237, December 2015. ISSN 1571-0661. doi: 10.1016/j.entcs.2015.12.014. URL http://dx.doi.org/10.1016/j.entcs.2015.12.014.

nLab. computational trinitarianism. https://ncatlab.org/nlab/show/computational+trinitarianism.

Jacques Carette, Chao-Hong Chen, Vikraman Choudhury, and Amr Sabry. From reversible programs to univalent universes and back. *arXiv preprint arXiv:1708.02710*, 2017.