Dagger Traced Symmetric Monoidal Categories and Reversible Programming

William J. Bowman, Roshan P. James, and Amr Sabry

School of Informatics and Computing, Indiana University
{wilbowma,rpjames,sabry}@indiana.edu

Abstract. We develop a reversible programming language from elementary mathematical and categorical foundations. The core language is based on isomorphisms between finite types: it is complete for combinational circuits and has an elegant semantics in dagger symmetric monoidal categories. The categorical semantics enables the definition of canonical and well-behaved reversible loop operators based on the notion of traced categories. The extended language can express recursive reversible algorithms on recursive types such as the natural numbers, lists, and trees. Computations in the extended language may diverge but every terminating computation is still reversible.

1 Introduction

The canonical computational models (e.g., Turing machines, λ-calculus, combinatory logic, etc.) are all based on irreversible processes. Hence it is not surprising that many attempts for developing reversible computational models start with an irreversible model and attempt to retrofit reversibility with a history mechanism [3, 4, 17].

In contrast, in Physics, the more fundamental notions describe processes in closed systems where every action is reversible. Open systems, which allow irreversible processes, are a derived notion — they can considered as a subsystem of a closed system. Following the approach pioneered by Toffoli [16] and used by some researchers [2, 6, 13, 18, 19], we adopt the physical idea that reversibility is the fundamental notion, and that irreversibility is a derived notion.

Structure of the paper and contributions. Our starting point, therefore, is that the design of a reversible language should be done from "first principles," not via any conventional irreversible model. The next section introduces the language Π based on elementary isomorphisms which are reversible by definition. These isomorphisms have an appealing categorical foundation and further allow us to transport other categorical constructions to a computational interpretation. Section 3 extends the language with a looping construct using the notion of traced categories. An implementation of the language along with several programming examples are available for download from `http://www.cs.indiana.edu/~sabry/papers/reversible.tar.gz`.

2 The Language Π of Isomorphisms

Starting with isomorphisms on simple arithmetic expressions, we develop a simple programming language that is universal for finite types. Although the language appears modest, it is universal for reversible combinational circuits.

2.1 Syntax and Type System

Consider the language of arithmetic expressions inductively built from the constants 0, 1, and addition and multiplication. The language admits the following sound and complete isomorphisms. (1) Addition and multiplication are commutative, associative, and have a neutral element. (2) Multiplication distributes over addition. We can turn the above system into a reversible programming language by simply viewing the arithmetic expressions as types and the isomorphisms as programs. When viewed as types, the arithmetic expressions are interpreted as follows. The type 0 is the empty type containing no inhabitants. The type 1 has exactly one inhabitant called (). The type $b_1 + b_2$ is the disjoint union of b_1 and b_2 whose elements are appropriately tagged values from either type. The type $b_1 * b_2$ is the type of ordered pairs whose elements are coming from b_1 and b_2 respectively. This interpretation can be formalized as follows:

$$\begin{aligned} \mathit{Values\ types}, b &::= 0 \mid 1 \mid b + b \mid b \times b \\ \mathit{Values}, v &::= () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v) \end{aligned}$$

The type system associates each value with its type:

$$\frac{}{\vdash () : 1} \quad \frac{\vdash v : b_1}{\vdash \mathit{left}\ v : b_1 + b_2} \quad \frac{\vdash v : b_2}{\vdash \mathit{right}\ v : b_1 + b_2} \quad \frac{\vdash v_1 : b_1 \quad \vdash v_2 : b_2}{\vdash (v_1, v_2) : b_1 \times b_2}$$

We now introduce *combinators* which are program constructs corresponding to the left-to-right and right-to-left reading of each isomorphism. These constructs will constitute the program building blocks in our language:

$$\begin{array}{rcl} \succcurlyeq_+: & 0 + b \leftrightarrow b & :\preccurlyeq_+ \\ \times_+ : & b_1 + b_2 \leftrightarrow b_2 + b_1 & : \times_+ \\ \gtrless_+: & b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3 & :\lessgtr_+ \\ \succcurlyeq_\times: & 1 \times b \leftrightarrow b & :\preccurlyeq_\times \\ \times_\times : & b_1 \times b_2 \leftrightarrow b_2 \times b_1 & : \times_\times \\ \gtrless_\times: & b_1 \times (b_2 \times b_3) \leftrightarrow (b_1 \times b_2) \times b_3 & :\lessgtr_\times \\ \prec_0 : & 0 \times b \leftrightarrow 0 & : \succ_0 \\ \multicolumn{3}{c}{\prec : (b_1 + b_2) \times b_3 \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3) : \succ} \end{array}$$

Each line of this table is to be read as the definition of two operators. For example corresponding to the 'identity of $\times$' isomorphism we have the two operators $\succcurlyeq_\times : 1 \times b \leftrightarrow b$ and $\preccurlyeq_\times : b \leftrightarrow 1 \times b$.

Now that we have primitive operators we need some means of composing them. We construct the composition combinators out of the closure conditions

for isomorphisms. Thus we have program constructs that witness reflexivity id, symmetry sym, and transitivity $\circ$ and two parallel composition combinators, one for sums $\oplus$ and one for pairs $\otimes$.

$$\frac{}{id : b \leftrightarrow b} \quad \frac{c : b_1 \leftrightarrow b_2}{sym\ c : b_2 \leftrightarrow b_1} \quad \frac{c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_2 \leftrightarrow b_3}{c_1 \circ c_2 : b_1 \leftrightarrow b_3}$$

$$\frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 \oplus c_2 : b_1 + b_2 \leftrightarrow b_3 + b_4} \quad \frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 \otimes c_2 : b_1 \times b_2 \leftrightarrow b_3 \times b_4}$$

2.2 Semantics

Every program construct has an adjoint that works in the other direction.

Proposition 1. *Each combinator $c : b_1 \leftrightarrow b_2$ has an adjoint $c^\dagger : b_2 \leftrightarrow b_1$.*

Given a program $c : b_1 \leftrightarrow b_2$ in Π, we can run it by supplying it with a value $v_1 : b_1$. The use of the sym constructor uses the adjoint to reverse the program. The evaluation rules $c\ v_1 \mapsto v_2$ are given below:

$$\begin{array}{ll}
\succcurlyeq_+ (right\ v) & \mapsto v \\
\preccurlyeq_+ v & \mapsto right\ v \\
\times_+ (left\ v) & \mapsto right\ v \\
\times_+ (right\ v) & \mapsto left\ v \\
\gtrdot_+ (left\ v_1) & \mapsto left\ (left\ v_1) \\
\gtrdot_+ (right\ (left\ v_2)) & \mapsto left\ (right\ v_2) \\
\gtrdot_+ (right\ (right\ v_3)) & \mapsto right\ v_3 \\
\lessdot_+ (left\ (left\ v_1)) & \mapsto left\ v_1 \\
\lessdot_+ (left\ (right\ v_2)) & \mapsto right\ (left\ v_2) \\
\lessdot_+ (right\ v_3) & \mapsto right\ (right\ v_3)
\end{array}
\qquad
\begin{array}{ll}
\succcurlyeq_\times ((), v) & \mapsto v \\
\preccurlyeq_\times v & \mapsto ((), v) \\
\times_\times (v_1, v_2) & \mapsto (v_2, v_1) \\
\gtrdot_\times (v_1, (v_2, v_3)) & \mapsto ((v_1, v_2), v_3) \\
\lessdot_\times ((v_1, v_2), v_3) & \mapsto (v_1, (v_2, v_3)) \\
\prec (left\ v_1, v_3) & \mapsto left\ (v_1, v_3) \\
\prec (right\ v_2, v_3) & \mapsto right\ (v_2, v_3) \\
\succ (left\ (v_1, v_3)) & \mapsto (left\ v_1, v_3) \\
\succ (right\ (v_2, v_3)) & \mapsto (right\ v_2, v_3)
\end{array}$$

Since there are no values that have the type 0, the reductions for the combinators $\succcurlyeq_+$, $\preccurlyeq_+$, $\prec_0$ and $\succ_0$ omit the impossible cases. The semantics of composition combinators is:

$$\frac{}{id\ v \mapsto v} \quad \frac{c^\dagger v_1 \mapsto v_2}{(sym\ c)\ v_1 \mapsto v_2} \quad \frac{c_1\ v_1 \mapsto v \quad c_2\ v \mapsto v_2}{(c_1 \circ c_2)\ v_1 \mapsto v_2}$$

$$\frac{c_1\ v_1 \mapsto v_2}{(c_1 \oplus c_2)\ (left\ v_1) \mapsto left\ v_2} \quad \frac{c_2\ v_1 \mapsto v_2}{(c_1 \oplus c_2)\ (right\ v_1) \mapsto right\ v_2}$$

$$\frac{c_1\ v_1 \mapsto v_3 \quad c_2\ v_2 \mapsto v_4}{(c_1 \otimes c_2)\ (v_1, v_2) \mapsto (v_3, v_4)}$$

We can now formalize that the adjoint of each construct c is its inverse in the sense that the evaluation of the adjoint maps the output of c to its input.

Proposition 2. Reversibility. *Iff $c\ v \mapsto v'$ then $c^\dagger v' \mapsto v$.*

The language Π has an elegant semantics in *dagger symmetric monoidal categories* (See Selinger's survey paper [15] for an excellent reference.) These categories constitute the backbone of reversible, quantum, and linear programming models [1].

Proposition 3. *The language Π can be interpreted as a dagger symmetric monoidal category in two ways: with $+$ as the monoidal tensor or with $\times$ as the monoidal tensor.*

3 Traced Categories and the Language Π^o

Traced categories, introduced by Joyal, Street and Verity [10], have proved useful for modeling recursion [7]. After defining these categories, we extend Π with the appropriate structures for recursion: isorecursive types and trace operators.

Definition 1. *A symmetric monoidal category with $+$ as the monoidal tensor, 0 as the identity, $\times_+$ as the symmetry natural transformation, is said to be* traced *if it is equipped with a family of functions, called a* trace*, such that for any morphism $f : (X + A) \to (X + B)$, we have a morphism $Tr^X_{A,B}(f) : A \to B$ subject to the following coherence conditions [8]:*

- ***Tightening:*** $Tr^X_{C,D}((id_X \oplus k) \circ f \circ (id_X \oplus h)) = k \circ Tr^X_{A,B}(f) \circ h$
- ***Yanking:*** $Tr^X_{X,X}(\times_+) = id_X = Tr^X_{X,X}(\times_+)$
- ***Superposing:*** $Tr^X_{A+C,B+C}(f \oplus id_C) = Tr^X_{A,B}(f) \oplus id_C$
- ***Exchange:***
  $Tr^X_{A,B}(Tr^Y_{X+A,X+B}(f)) = Tr^Y_{A,B}(Tr^X_{Y+A,Y+B}((\times_+ \oplus id_B) \circ f \circ (\times_+ \oplus id_A)))$

3.1 Recursive Types

We begin by extending Π with *isorecursive types*. These types can express all the usual inductive structures, like the natural numbers, lists, and trees. In addition, they naturally fit within the framework of reversible languages as they come equipped with two isomorphisms *fold* and *unfold* that witness the equivalence of a value of a recursive type with all its "unrollings." In more detail, we extend the type, values and isomorphisms as follows:

$$b ::= \ldots \mid x \mid \mu x.b \qquad v ::= \ldots \mid \langle v \rangle \qquad \mathit{fold} : b[\mu x.b/x] \leftrightarrow \mu x.b : \mathit{unfold} \qquad \frac{\vdash v : b[\mu x.b/x]}{\vdash \langle v \rangle : \mu x.b}$$

In the type $\mu x.b$, the type b typically includes occurrences of x that (recursively) refer back to $\mu x.b$. This isomorphism between the occurrences of x and $\mu x.b$ is witnessed by the two combinators *fold* and *unfold*. To create recursive values, we introduce the notation $\langle v \rangle$ that allows the construction of arbitrarily large values of a given recursive type. Intuitively, to construct a value of type $\mu x.b$, we must have a value of type $b[\mu x.b/x]$. Depending on the structure of b, this may or not be possible. For example, if the recursive type is $\mu x.x$ then to construct a value of that type, we need to have a value of the same type, ad

infinitum. In contrast, if the recursive type is $\mu x.1 + x$, then we can create the initial value *left* () of type $1+(\mu x.1+x)$ which leads to the value $\langle \mathit{left}\ () \rangle$ of type $\mu x.1 + x$ and then $\langle \mathit{right}\, \langle \mathit{left}\ () \rangle \rangle$, $\langle \mathit{right}\, \langle \mathit{right}\, \langle \mathit{left}\ () \rangle \rangle \rangle$ and so on. In fact the type $\mu x.1 + x$ represents the natural numbers in unary format. The semantics of *fold* and *unfold* is simply $\mathit{fold}\ v \mapsto \langle v \rangle$ and $\mathit{unfold}\, \langle v \rangle \mapsto v$.

3.2 Trace Operators

The definition of traced categories is expressed using a particular monoidal tensor. We have two possible tensors in Π which in principle can be extended to support trace operators. The natural tensor to use in this case, however, is the sum operator + as it leads to usual looping constructs as explained next. Specifically, we extend Π with a trace operator that has this typing rule:

$$\frac{c : b_1 + b_2 \leftrightarrow b_1 + b_3}{\mathit{trace}\ c : b_2 \leftrightarrow b_3}$$

Intuitively, we are given a computation c that accepts a value of type $b_1 + b_2$ and we build a looping version *trace* c that only takes a value of type b_2 and evaluates as follows. The value of type b_2 is injected into the sum type $b_1 + b_2$ by tagging it with the *right* constructor and passing it to c. As long as c returns a value that is tagged with *left*, that value is fed back to c. As soon as a value tagged with *right* is returned, that value is returned as the final answer of the *trace* c computation. Formally, we can express this semantics as follows:

$$\frac{(c \circ \mathit{loop}_c)\ (\mathit{right}\ v_1) \mapsto v_2}{(\mathit{trace}\ c)\ v_1 \mapsto v_2} \quad \frac{(c \circ \mathit{loop}_c)\ (\mathit{left}\ v_1) \mapsto v_2}{\mathit{loop}_c\ (\mathit{left}\ v_1) \mapsto v_2} \quad \frac{}{\mathit{loop}_c\ (\mathit{right}\ v) \mapsto v}$$

where loop_c is an internal combinator. We note that the evaluation *trace* c may diverge.

We now establish that Π^o retains all the properties of Π:

Proposition 4. *Each combinator $c : b_1 \leftrightarrow b_2$ has an adjoint $c^\dagger : b_2 \leftrightarrow b_1$.*

Proposition 5. Reversibility of Π^o. *Iff $c\ v \mapsto v'$ then $c^\dagger v' \mapsto v$.*

Proposition 6. *The language Π^o can be interpreted as a traced category with + as the monoidal tensor.*

4 Conclusion

We have presented Π and its extension Π^o which are languages that have clear categorical connections and every computation is inherently reversible. We provide many programming examples in the code accompanying the paper. Finally, we briefly outline connections with other related work.

Several papers achieve reversibility via compilation [12, 5] whereas we propose a model of computation, Π^o, that is inherently reversible. Proposals for

reversible computation such as rCL [14], SECD-H [9], SE(M)CD [11] and Pendulum [18] consist of an irreversible core and some form of "history tracking" for reverse execution. Π^o does not rely on any such history tracking. The Janus language [19] is reversible by design, but requires the programmer to divine correct 'exit predicates' for backward determinism, unlike Π^o where all expressible computations are reversible. In spirit, Π^o is much like Abramsky's biorthogonal automata [1] which lack a term language - but in principle could be given one.

Acknowledgments. We thank Erik Wennstrom for feedback on trace operators in category theory and the anonymous reviewers for their excellent feedback and corrections.

References

1. Abramsky, S.: A structural approach to reversible computation. Theor. Comput. Sci. 347, 441–464 (December 2005)
2. Axelsen, H.B., Glück, R.: What do reversible programs compute? In: FOSSACS. pp. 42–56 (2011)
3. Bennett, C.: Logical reversibility of computation. IBM J. Res. Dev. 17 (1973)
4. Di Pierro, A., Hankin, C., Wiklicky, H.: Reversible combinatory logic. Mathematical. Structures in Comp. Sci. 16, 621–637 (August 2006)
5. Glück, R., Kawabe, M.: Revisiting an automatic program inverter for lisp. SIGPLAN Notices 40(5), 8–17 (2005)
6. Green, A.S., Altenkirch, T.: From reversible to irreversible computations. Electron. Notes Theor. Comput. Sci. 210, 65–74 (July 2008)
7. Hasegawa, M.: Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In: TLCA. pp. 196–213. Springer-Verlag (1997)
8. Hasegawa, M.: On traced monoidal closed categories. Mathematical. Structures in Comp. Sci. 19, 217–244 (April 2009)
9. Huelsbergen, L.: A logically reversible evaluator for the call-by-name lambda calculus. InterJournal Complex Systems 46 (1996)
10. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. In: Mathematical Proceedings of the Cambridge Philosophical Society. Cambridge Univ Press (1996)
11. Kluge, W.E.: A Reversible SE(M)CD Machine. In: IFL. pp. 95–113 (1999)
12. Mu, S.C., Bird, R.S.: Inverting functions as folds. In: MPC. pp. 209–232 (2002)
13. Mu, S.C., Hu, Z., Takeichi, M.: An injective language for reversible computation. In: Mathematics of Program Construction. pp. 289–313 (2004)
14. Pierro, A.D., Hankin, C., Wiklicky, H.: Reversible combinatory logic. Mathematical Structures in Computer Science 16(4), 621–637 (2006)
15. Selinger, P.: A survey of graphical languages for monoidal categories. In: Coecke, B. (ed.) New Structures for Physics, Lecture Notes in Physics, vol. 813, pp. 289–355. Springer Berlin / Heidelberg (2011)
16. Toffoli, T.: Reversible computing. In: Automata, Languages and Programming. pp. 632–644. Springer-Verlag (1980)
17. Van Tonder, A.: A lambda calculus for quantum computation. SIAM J. Comput. 33, 1109–1135 (May 2004)
18. Vieri, C.J.: Pendulum: A Reversible Computer Architecture. Master's thesis, MIT (May 1995)
19. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: PEPM'07. pp. 144–153. ACM (2007)