Isomorphic Interpreters from Logically Reversible Abstract Machines

Roshan P. James and Amr Sabry

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

Abstract. In our previous work, we developed a reversible programming language and established that every computation in it is a (partial) isomorphism that is reversible and that preserves information. The language is founded on type isomorphisms that have a clear categorical semantics but that are awkward as a notation for writing actual programs, especially recursive ones. This paper remedies this aspect by presenting a systematic technique for developing a large and expressive class of reversible recursive programs, that of logically reversible small-step abstract machines. In other words, this paper shows that once we have a logically reversible machine in a notation of our choice, expressing the machine as an isomorphic interpreter can be done systematically and does not present any significant conceptual difficulties. Concretely, we develop several simple interpreters over numbers and addition, move on to tree traversals, and finish with a meta-circular interpreter for our reversible language. This gives us a means of developing large reversible programs with the ease of reasoning at the level of a conventional small-step semantics.

1 Introduction

In recent papers [3, 7], we developed a pure reversible model of computation that is obtained from the type isomorphisms and categorical structures that underlie models of linear logic and quantum computing and that treats information as a linear resource that can neither be erased nor duplicated. From a programming perspective, our model gives rise to a pure (with no embedded computational effects such as assignments) reversible programming language Π^o based on *partial isomorphisms*. In more detail, in the recursion-free fragment of Π^o, every program witnesses an isomorphism between two finite types; in the full language with recursion, some of these isomorphisms may be partial, i.e., may diverge on some inputs.

In this paper, we investigate the practicality of writing large recursive programs in Π^o. Specifically, we assume that we are given some reversible recursive algorithm expressed as a *reversible abstract machine*, and we show via a number of systematic steps, how to develop a Π^o program from that abstract machine. Our choice of starting from reversible abstract machines is supported by the following observations:

– it is an interesting enough class of reversible programs: researchers have invested the effort in manually designing such machines, e.g., the SE(M)CD machine [8], and the SECD-H [6];
– every reversible interpreter can be realized as such a machine: this means that our class of programs includes self-interpreters which are arguably a measure of the strength of any reversible language [2, 12, 13];
– general recursive programs can be systematically transformed to abstract machines: the technique is independent of reversible programs and consists of transforming general recursion to tail recursion and then applying *fission* to split the program into a driver and a small-step machine [5, 9].

To summarize, we assume we are given some reversible abstract machine and we show how to derive a Π^o program that implements the semantics of the machine. Our derivation is systematic and expressive. In particular, Π^o can handle machines with stuck states because it is based on partial isomorphisms. We illustrate our techniques with simple machines that do bounded iteration on numbers, tree traversals, and a meta-circular interpreter for Π^o.

2 Review of the Reversible Language: Π^o

We briefly review the reversible language Π^o introduced in our previous work [3, 7]. The set of types includes the empty type 0, the unit type 1, sum types $b_1 + b_2$, product types $b_1 \times b_2$, and recursive types $\mu x.b$. The set of values v includes () which is the only value of type 1, *left* v and *right* v which inject v into a sum type, (v_1, v_2) which builds a value of product type, and $\langle v \rangle$ which is used to construct recursive values. There are no values of type 0:

$$\begin{aligned} \mathit{value\ types}, b &::= 0 \mid 1 \mid b + b \mid b \times b \mid x \mid \mu x.b \\ \mathit{values}, v &::= () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v) \mid \langle v \rangle \end{aligned}$$

The expressions of Π^o are witnesses to the following type isomorphisms:

$$\begin{array}{rcl} \mathit{zeroe}: & 0 + b \leftrightarrow b & : \mathit{zeroi} \\ \mathit{swap}^+: & b_1 + b_2 \leftrightarrow b_2 + b_1 & : \mathit{swap}^+ \\ \mathit{assocl}^+: & b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3 & : \mathit{assocr}^+ \\ \mathit{unite}: & 1 \times b \leftrightarrow b & : \mathit{uniti} \\ \mathit{swap}^\times: & b_1 \times b_2 \leftrightarrow b_2 \times b_1 & : \mathit{swap}^\times \\ \mathit{assocl}^\times: & b_1 \times (b_2 \times b_3) \leftrightarrow (b_1 \times b_2) \times b_3 & : \mathit{assocr}^\times \\ \mathit{distrib}_0: & 0 \times b \leftrightarrow 0 & : \mathit{factor}_0 \\ \mathit{distrib}: & (b_1 + b_2) \times b_3 \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3) & : \mathit{factor} \\ \mathit{fold}: & b[\mu x.b/x] \leftrightarrow \mu x.b & : \mathit{unfold} \end{array}$$

Each line of the above table introduces one or two combinators that witness the isomorphism in the middle. Collectively the isomorphisms state that the structure $(b, +, 0, \times, 1)$ is a *commutative semiring*, i.e., that each of $(b, +, 0)$ and $(b, \times, 1)$ is a commutative monoid and that multiplication distributes over addition. The last isomorphism witnesses the equivalence of a value of recursive type

with all its “unrollings.” The isomorphisms are extended to form a congruence relation by adding the following constructors that witness equivalence and compatible closure. In addition, the language includes a *trace* operator to express looping:

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

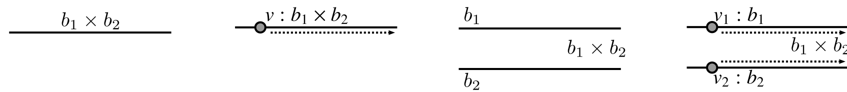
$$\frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 + c_2 : b_1 + b_2 \leftrightarrow b_3 + b_4} \quad \frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 \times c_2 : b_1 \times b_2 \leftrightarrow b_3 \times b_4} \quad \frac{c : b_1 + b_2 \leftrightarrow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

Adjoint. An important property of the language is that every combinator c has an adjoint $c^\dagger$ that reverses the action of c. This is evident by construction for the primitive isomorphisms. For the closure combinators, the adjoint is homomorphic except for the case of sequencing in which the order is reversed, i.e., $(c_1 \fatsemi c_2)^\dagger = (c_2{}^\dagger) \fatsemi (c_1{}^\dagger)$.

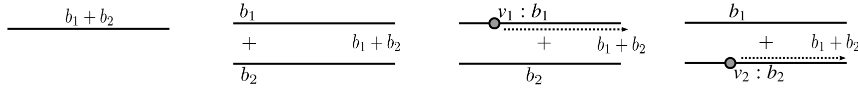
Graphical Language. Following the tradition for computations in monoidal categories [10], we present a graphical notation that conveys the semantics of Π^o. The general idea of the graphical notation is that combinators are modeled by “wiring diagrams” or “circuits” and that values are modeled as “particles” or “waves” that may appear on the wires. Evaluation therefore is modeled by the flow of waves and particles along the wires.

– The simplest sort of diagram is the $id : b \leftrightarrow b$ combinator which is simply represented as a wire labeled by its type b, as shown below on the left. In more complex diagrams, if the type of a wire is obvious from the context, it may be omitted. When tracing a computation, one might imagine a value v of type b on the wire, as shown below on the right:

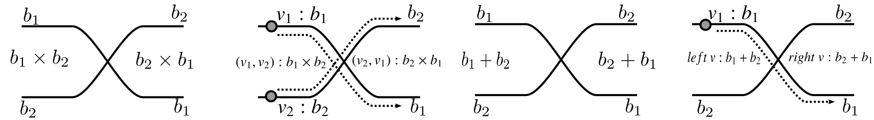
– The product type $b_1 \times b_2$ may be represented using either one wire labeled $b_1 \times b_2$ or two parallel wires labeled b_1 and b_2. In the case of products represented by a pair of wires, when tracing execution using particles, one should think of one particle on each wire or alternatively as in folklore in the literature on monoidal categories as a “wave:”



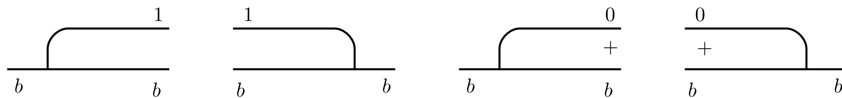
– Sum types may similarly be represented by one wire or using parallel wires with a + operator between them. When tracing the execution of two additive wires, a value can reside on only one of the two wires:



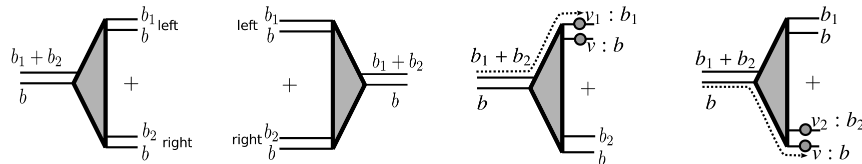
- Associativity is implicit in the graphical language. Thus, three parallel wires may represent $b_1 \times (b_2 \times b_3)$ or $(b_1 \times b_2) \times b_3$.
- Commutativity is represented by crisscrossing wires:



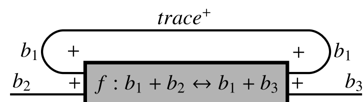
- The morphisms that witness that 0 and 1 are the additive and multiplicative units are represented as shown below. Note that since there is no value of type 0, there can be no particle on a wire of type 0. Also since the monoidal units can be freely introduced and eliminated, in many diagrams they are omitted and dealt with explicitly only when they are of special interest:



- Distributivity and factoring are interesting because they represent interactions between sum and pair types. Distributivity should essentially be thought of as a multiplexer that redirects the flow of $v : b$ depending on what value inhabits the type $b_1 + b_2$, as shown below. Factoring is the corresponding adjoint operation:



- Operations *fold* and *unfold* are specific to each recursive data type and are drawn as triangular boxes. For instance, Sec. 2.2 shows *fold*/*unfold* isomorphisms for numbers, $nat = \mu x.(1 + x)$.
- The *trace* operation is drawn as a looped circuit, where the traced type b_1 is shown as flowing backwards:

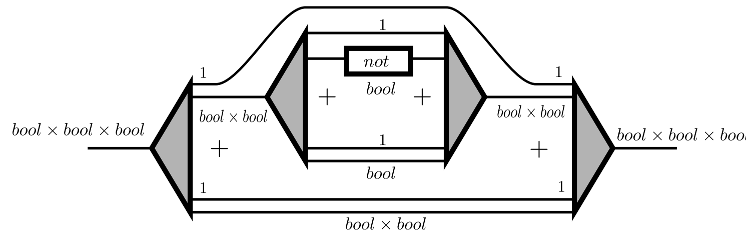


2.1 Universality.

As an example, consider the implementation of the Toffoli gate below, which establishes that Π^o is complete for combinational circuits. The Toffoli gate takes three boolean inputs: if the first two inputs are *true* then the third is negated. This encoding uses the type $1 + 1$ as *bool* and values *left* () and *right*() as *true* and *false*. Boolean negation, $not : bool \leftrightarrow bool$, is simply $swap^{+}$.

Even though Π^o lacks conditional expressions, they are expressible using the distributivity laws. Given any combinator $c : b \leftrightarrow b$, we can construct a combinator called $if_c : bool \times b \leftrightarrow bool \times b$ in terms of c, which behaves like a one-armed if-expression. If the supplied boolean is *true* then the combinator c is used to transform the value of type b. If the boolean is *false* then the value of type b remains unchanged. We can write down the combinator for if_c in terms of c as $distrib \fatsemi ((id \times c) + id) \fatsemi factor$.

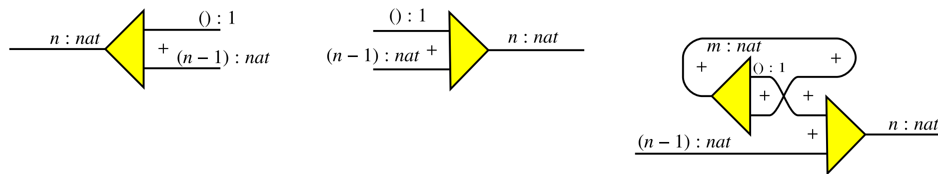
Given if_c, constructing the Toffoli gate is straightforward. If we choose c to be *not* we get if_{not} which is controlled-not, $cnot : bool \times bool \leftrightarrow bool \times bool$. Further, if_{cnot} is the required Toffoli gate, $toffoli : bool \times (bool \times bool) \leftrightarrow bool \times (bool \times bool)$, and is drawn below.



Previous work [7, Sec. 5] establishes that Π^o is Turing complete by showing the compilation of a Turing complete language — a first-order language with numbers and iteration — to Π^o.

2.2 Numeric Operations.

We encode numbers, $nat = 0 \mid n + 1$, using the recursive Π^o type $\mu x.1 + x$. The *fold*/*unfold* isomorphisms for *nat* are $unfold : nat \leftrightarrow 1 + nat : fold$.



The combinator on the left denotes the *unfold* isomorphism, which works as follows: If the number n is zero, the output is the top branch which has type 1. If the number is non-zero, the output is on the bottom branch and has value $n - 1$. The combinator in the middle is its dual *fold* isomorphism.

As explained in previous work [7, Sec. 4.2], numeric addition and subtraction can be expressed in Π^o as partial isomorphisms: The combinator on the right, $add_1 : nat \leftrightarrow nat$, acts like the successor function returning $n+1$ for all inputs n. Its adjoint, sub_1, obtained by flipping the diagram right to left, is a partial map that diverges on input 0 and returns $n-1$ for all other inputs n.

3 Simple Bounded Number Iteration

We illustrate the main concepts and constructions using two simple examples that essentially count n steps. The first machine does nothing else; the second uses this counting ability to add two numbers.

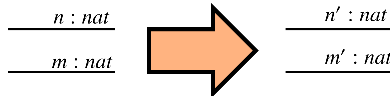
3.1 Counting

The first machine is defined as follows:

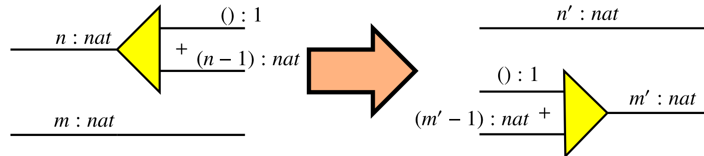
$$\begin{array}{rlrl} \mathit{Numbers}, n, m = & 0 \mid n+1 & \mathit{Start\ state} = & \langle n, 0 \rangle \\ \mathit{Machine\ states} = & \langle n, n \rangle & \mathit{Stop\ State} = & \langle 0, n \rangle \end{array}$$
$$\langle n+1, m \rangle \mapsto \langle n, m+1 \rangle$$

The machine is started with a number n in the first position and 0 in the second. Each reduction step decrements the first number and increments the second. The machine stops when the first number reaches 0, thereby taking exactly n steps. For example, if the machine is started in the configuration $\langle 3, 0 \rangle$, it would take exactly 3 steps to reach the final configuration: $\langle 3, 0 \rangle \mapsto \langle 2, 1 \rangle \mapsto \langle 1, 2 \rangle \mapsto \langle 0, 3 \rangle$. Dually, since the machine transition is clearly reversible, the machine can execute backwards from the final configuration to reach the initial configuration in also 3 steps.

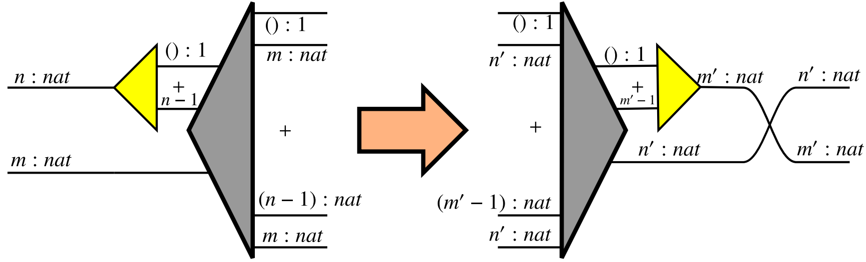
Our goal is to implement this abstract machine in Π^o. We start by writing a combinator $c : nat \times nat \leftrightarrow nat \times nat$ that executes one step of the machine transition. The combinator, when iterated, should map $(n, 0)$ to $(0, n)$ by precisely mimicking the steps of the abstract machine. Let us analyze this desired combinator c in detail starting from its interface:



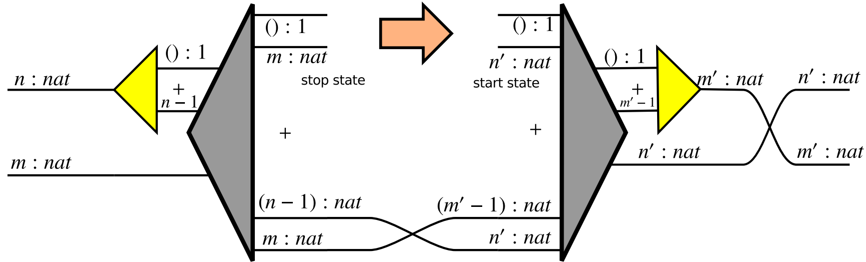
The reduction step of the machine examines the first nat and reconstructs the second nat. In other words, the first nat is unfolded to examine its structure and the second nat is folded to reconstruct it:



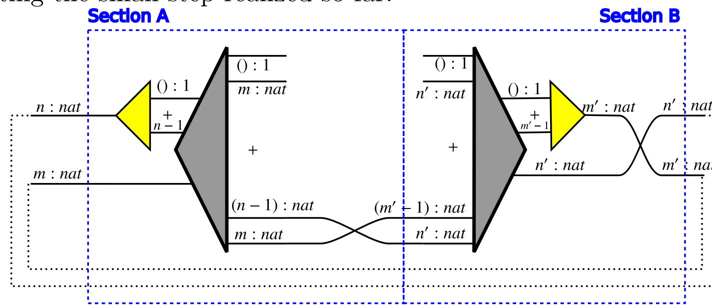
We now use distribution to isolate each possible machine state:



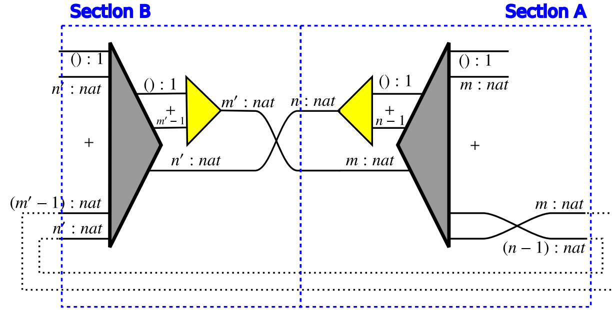
At this point, it is clear that the machine's transition consists connecting the $n-1$ input wire to n' on the output side and the m input wire to $m'-1$ on the output side:



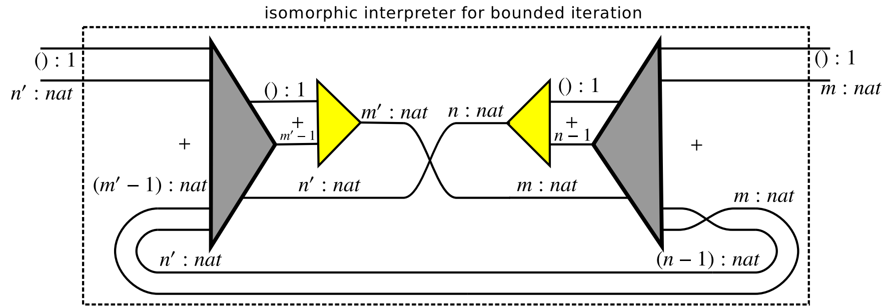
We are now close to the completion of the interpreter. The branch labeled $((), m)$ corresponds to the machine state $\langle 0, m\rangle$ which is the stop state of the machine. Similarly the branch labeled $((), n')$ corresponds to the start state of the machine. The last step in the construction involves introducing a *trace* operation for iterating the small step realized so far:



Sliding sections A and B past each other while maintaining the connectivity of the wires, exposes that the middle connections were really recursive calls to the machine transition:



The completed interpreter is given below:



3.2 Steps of the Construction.

Although trivial, the previous example captures the fundamental aspects of our construction. In general, our starting point is an abstract machine in which every rewrite step is logically reversible. The formal criterion of logical reversibility in this setting is captured by Abramsky's *bi-orthogonal automata* [1, Sec. 2] which is summarized as:

1. Machine states are described as tuples whose components are algebraic data types. In the above example we used only the *nat* data type and tuples of the form $\langle n, m \rangle$.
2. Machines must have distinguishable start and stop states. There may be more than one valid start state and more than one valid stop state, however.
3. Each valid machine state must match a unique pattern on the left-hand side and right-hand side of the '$\mapsto$' relation.
4. Every reduction step must be computable (in Π^o), must not drop or duplicate pattern variables and must be logically reversible — i.e. it must be possible to reduce from right to left.

Given such a machine the process of encoding the machine in Π^o consists of the following steps:

1. Expand the input to expose enough structure to distinguish the left-hand side of each machine state;

2. Expand the output to expose enough structure to distinguish the right-hand side of each machine state;
3. Shuffle matching input terms to output terms, inserting any appropriate mediating computations.
4. Slide the two sections to expose the start and stop state and introduce a *trace* to iterate the construction.

3.3 Adder

Let us apply these steps to the slightly more interesting example of an adder:

$$\begin{array}{rlrl} \mathit{Numbers}, n, m, p = & 0 \mid n+1 & \mathit{Start\ state} = & \langle n, n, 0\rangle \\ \mathit{Machine\ states} = & \langle n, n, n\rangle & \mathit{Stop\ State} = & \langle 0, n, n\rangle \end{array}$$

$$\langle n+1, p, m\rangle \mapsto \langle n, p+1, m+1\rangle$$

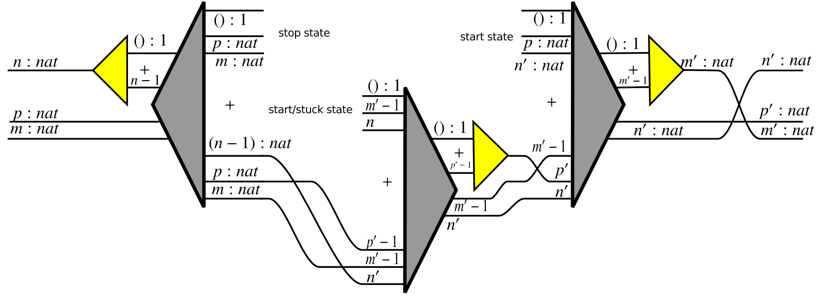
The idea of the machine is to start with 3 numbers: the two numbers to add and an accumulator initialized to 0. Each step of the machine, decrements one of the numbers and increments the second number and the accumulator. For example, $\langle 3,4,0\rangle \mapsto \langle 2,5,1\rangle \mapsto \langle 1,6,2\rangle \mapsto \langle 0,7,3\rangle$. In general, the sum of the two numbers will be in the second component, and the last component is supposed to record enough information to make the machine reversible.

A closer look however reveals that the machine defines a *partial* isomorphism: not all valid final states can be mapped to valid start states. Indeed consider the configuration $\langle 0,2,3\rangle$ which is a valid final state. Going backwards, the transitions start as follows $\langle 0,2,3\rangle \mapsto \langle 1,1,2\rangle \mapsto \langle 2,0,1\rangle$ at which point, the machine gets stuck at a state that is not a valid start state. This is a general problem that we discuss below in detail.

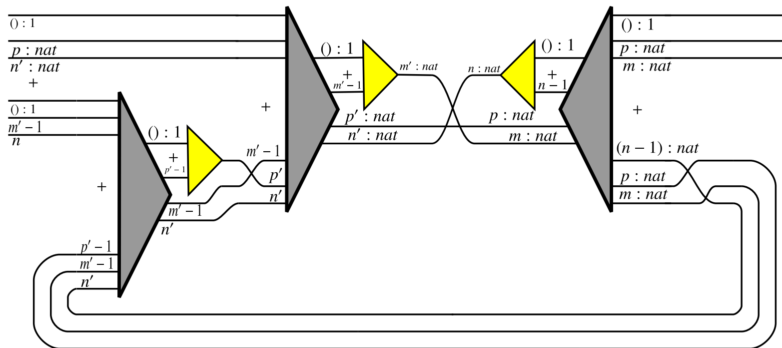
Stuck States. The type systems of most languages are not expressive enough to encode the precise domain and range of a function. For example, in most typed languages, division by zero is considered type-correct and the runtime system is required to deal with such an error. A stuck state of an abstract machine is just a manifestation of this general problem. The common solutions are:

- *Use a more expressive type system.* One could augment Π^o with a richer type system that distinguishes non-zero numbers from those that can be zero, thereby eliminating the sub_1 0 situation entirely. Similarly, in the meta-circular interpreter in Sec. 4.3, one could use generalized abstract data types (GADTs [4, 11]) to eliminate the stuck states.
- *Diverge.* Another standard approach in dealing with stuck states is to make the machine diverge or leave the output undefined or unobservable in some way. If the specific case of the machine above, we can use the primitive add_1 whose dual sub_1 is undefined when applied to 0 (see Sec. 2.2).

– *Stop the machine.* Alternatively, we can consider the stuck state as a valid final state. In the case of the example above, we would treat states of the form $\langle n, 0, n\rangle$ as valid stop states for reverse execution. This gives us two valid start states in the case of forward execution and makes the isomorphism total. If we chose this approach, the machine would look as shown below by the end of *step* 3:



Note that there are indeed two "start states" in the interpreter. As before, we can slide the two sides of the diagram and tie the knot using *trace* to get the desired interpreter:



4 Advanced Examples

We show the generality of our construction by applying it to three non-trivial examples: a tree traversal, parity translation of numbers and a meta-circular interpreter for Π^o.

4.1 Tree Traversal

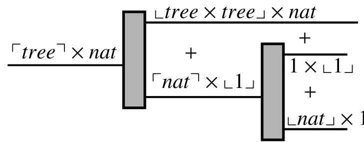
The type of binary trees we use is $\mu x.(nat + x \times x)$, i.e., binary trees with no information at the nodes and with natural numbers at the leaves. To define the abstract machine, we need a notion of *tree contexts* to track which subtree is currently being explored. The definitions are shown on the left:

$$
\begin{array}{rl}
\mathit{Tree}, t = & L\ n \mid N\ t\ t \\
\mathit{Tree\ Contexts}, c = & \Box \mid \mathit{Lft}\ c\ t \mid \mathit{Rgt}\ t\ c \\
\mathit{Machine\ states} = & \langle t, c \rangle \mid \{c, t\} \\
\mathit{Start\ state} = & \langle t, \Box \rangle \\
\mathit{Stop\ State} = & \{\Box, t\}
\end{array}
\qquad
\begin{array}{rcl}
\langle L\ n, c \rangle & \mapsto & \{c, L\ (n+1)\} \\
\langle N\ t_1\ t_2, c \rangle & \mapsto & \langle t_1, \mathit{Lft}\ c\ t_2 \rangle \\
\{\mathit{Lft}\ c\ t_2, t_1\} & \mapsto & \langle t_2, \mathit{Rgt}\ t_1\ c \rangle \\
\{\mathit{Rgt}\ t_1\ c, t_2\} & \mapsto & \{N\ t_1\ t_2, c\}
\end{array}
$$

The reduction rules on the right traverse a given tree and increment every leaf value. The machine here is a little richer than the ones dealt with previously. In particular, we have two types of machine states $\langle t, c \rangle$ and $\{t, c\}$. The first of these corresponds to walking down a tree, building up the context in the process. The second corresponds to reconstructing the tree from the context and also switching focus to any unexplored subtrees in the process. There are also two syntactic categories to deal with (trees and tree contexts) where previously we only had numbers. The *fold* and *unfold* isomorphisms that we need for trees and tree contexts are:

$$
\begin{array}{rcl}
\mathit{unfold}_t : & t \leftrightarrow n + t \times t & : \mathit{fold}_t \\
\mathit{unfold}_c : & c \leftrightarrow 1 + c \times t + t \times c & : \mathit{fold}_c
\end{array}
$$

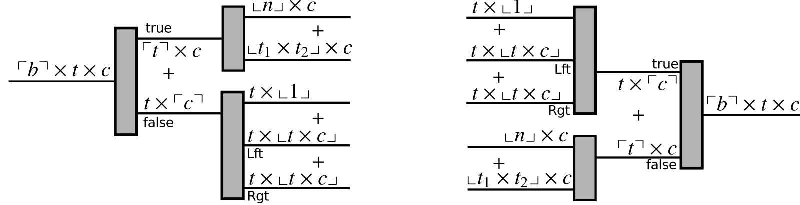
New Notation. To make the diagrams easier to understand, we introduce a syntactic convenience which combines the first steps of the construction that consist of *fold* / *unfold* and *distribute* / *factor*. We collectively represent these steps using thin vertical rectangle. Also we will introduce the convention that the component that is being expanded (or constructed) will be marked by using $\ulcorner\urcorner$ and the components that are being generated (or consumed) will be marked by $\llcorner\lrcorner$. For example, given a value of type $\mathit{tree} \times \mathit{nat}$, the diagram below shows how to first expand the *tree* component and then in one of the generated branches, expand the *nat* component:



We can now apply our construction. The first step to developing the isomorphic interpreter is to recognize that the two possible kinds of machine states simply hide an implicit *bool*. We make this explicit:

$$
\begin{array}{rl}
\mathit{Machine\ states} = & \langle \mathit{bool}, t, c \rangle \\
\mathit{Start\ state} = & \langle \mathit{true}, t, \Box \rangle \\
\mathit{Stop\ state} = & \langle \mathit{false}, t, \Box \rangle
\end{array}
\qquad
\begin{array}{rcl}
\langle \mathit{true}, L\ n, c \rangle & \mapsto & \langle \mathit{false}, L\ (n+1), c \rangle \\
\langle \mathit{true}, N\ t_1\ t_2, c \rangle & \mapsto & \langle \mathit{true}, t_1, \mathit{Lft}\ c\ t_2 \rangle \\
\langle \mathit{false}, t_1, \mathit{Lft}\ c\ t_2 \rangle & \mapsto & \langle \mathit{true}, t_2, \mathit{Rgt}\ t_1\ c \rangle \\
\langle \mathit{false}, t_2, \mathit{Rgt}\ t_1\ c \rangle & \mapsto & \langle \mathit{false}, c, N\ t_1\ t_2 \rangle
\end{array}
$$

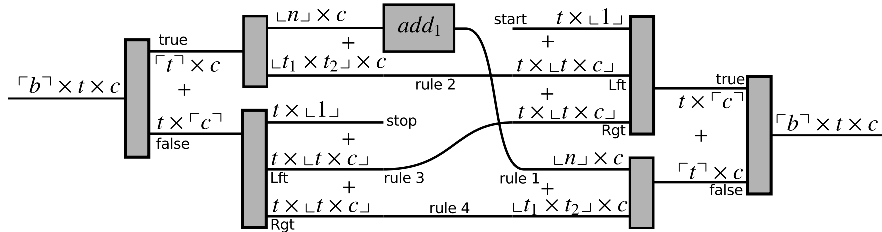
We start examining the machine components as before:



On the input side, for *true* states, we have expanded the tree component and for *false* states we have expanded the tree contexts. We have done the opposite on the output side exactly matching up what the abstract machine does. One thing to note is that we dropped the 1 introduced by expanding the *bool* and instead just labeled the *true* and *false* branches.

We are ready to start connecting the machine states corresponding to the reductions that we would like:

1. When we encounter a leaf we would like to increment its value and move to the corresponding *false* machine state. For the sake of simplicity, in this interpreter we won't be concerned with exposing stuck states: we simply use add_1 whose adjoint sub_1 diverges when applied to 0.
2. For all the other reduction rules, it is a straightforward mapping of related states following the reduction rules. For readability, we have annotated the diagram below with the names of the reduction rules and we have included subscripts on the various ts to indicate any implicit swaps that should be inserted.



This essentially completes the construction of the (partially) isomorphic tree-traversal interpreter, except for the final step which slides the input and output sides.

4.2 Parity Translation

Deriving Π^o combinators from abstract machines can sometimes be used as an efficient indirect way to program in Π^o. It is well known that every number n can be represented as $2a+0$ or $2a+1$, depending on whether it is odd or even. The later can be represented by the algebraic data type $par = odd \mid even \mid A\ par$

where the nesting of A constructor indicates the value of a. For example $0 = even, 1 = odd, 2 = A\ even, 3 = A\ odd, 4 = A\ (A\ even)$ etc.

Say we wish to build a Π^o combinator to map *nat* to its parity encoded version *par*. While the type *par* is expressible as $\mu x.(1+1)+x$ in Π^o, it is not a fixed unfolding of $nat = \mu x.1+x$ and hence it is not immediately apparent how such a combinator can be constructed.

Instead of directly programming in Π^o however, one can derive the required $nat \leftrightarrow par$ combinator by first constructing an abstract machine that maps *nat* to *par* and then translating it to Π^o. We first express *nat* and *par* algebraically and then develop a logically reversible abstract machine:

$$\begin{aligned} Numbers, n, m &= 0 \mid n+1 \\ Parity, par &= even \mid odd \mid A\ par \end{aligned}$$

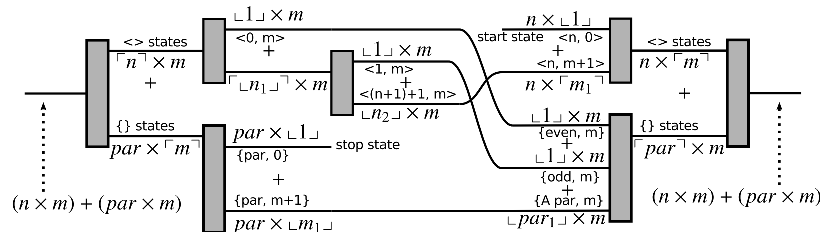
Machine States:

$$\begin{aligned} Machine\ states &= \langle nat, nat \rangle \ \mid \{parity, nat\} \\ Start\ state &= \langle n, 0 \rangle \\ Stop\ state &= \{par, 0\} \end{aligned}$$

Machine Reductions:

$$\begin{aligned} \langle 0, m \rangle &\mapsto \{even, m\} \\ \langle 1, m \rangle &\mapsto \{odd, m\} \\ \langle (n+1)+1, m \rangle &\mapsto \langle n, m+1 \rangle \\ \{par, m+1\} &\mapsto \{A\ par, m\} \end{aligned}$$

The derivation of the Π^o combinator is straightforward and proceeds as before. The partial combinator representing the wiring of machine reductions is shown below.



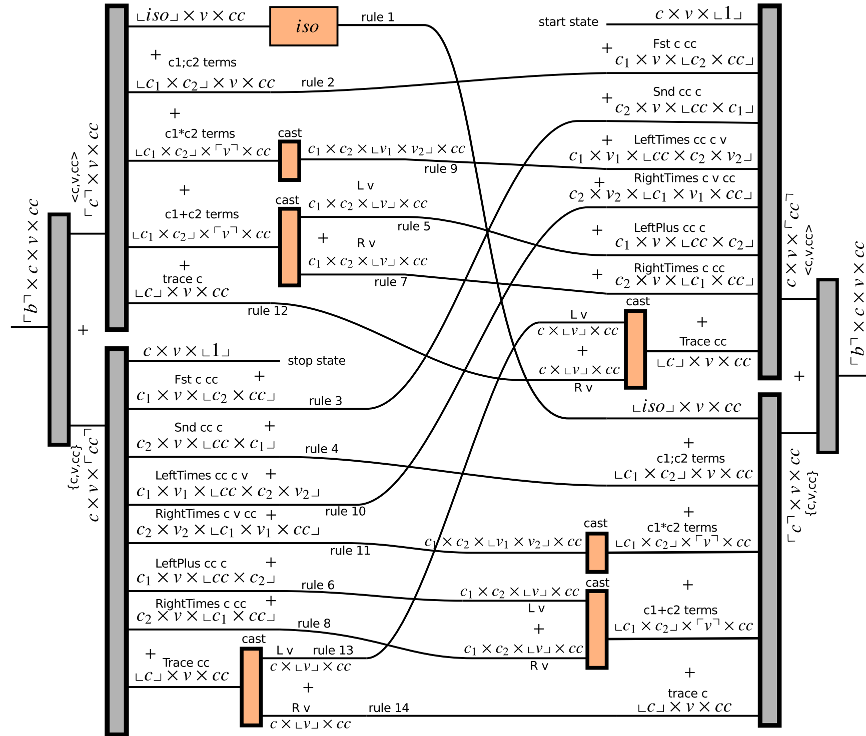
We thank Fritz Henglein for the motivation underlying this example.

4.3 A Π^o Interpreter

In the final construction we present a meta-circular interpreter for Π^o written in Π^o. This is a a non-trivial abstract machine with several cases, but the derivation follows in exactly the same way as before. Here is a logically reversible small-step abstract machine for Π^o and the derived isomorphic interpreter with the reductions labeled.

$$
\begin{array}{rcl}
\mathit{Combinators}, c & = & \mathit{iso} \mid c \fatsemi c \mid c \times c \mid c + c \mid \mathit{trace}\ c \\
\mathit{Combinator\ Contexts}, cc & = & \Box \mid \mathit{Fst}\ cc\ c \mid \mathit{Snd}\ c\ cc \\
 & \mid & \mathit{LeftTimes}\ cc\ c\ v \mid \mathit{RightTimes}\ c\ v\ cc \\
 & \mid & \mathit{LeftPlus}\ cc\ c \mid \mathit{RightPlus}\ c\ cc \mid \mathit{Trace}\ cc \\
\mathit{Values}, v & = & () \mid (v, v) \mid L\ v \mid R\ v
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Machine\ states} & = & \langle c, v, cc\rangle \mid \{c, v, cc\} & \\
\mathit{Start\ state} & = & \langle c, v, \Box\rangle & \\
\mathit{Stop\ State} & = & \{c, v, \Box\} & \\
\langle \mathit{iso}, v, cc\rangle & \mapsto & \{\mathit{iso}, \mathit{iso}(v), cc\} & \mathit{rule}\ 1 \\
\langle c_1 \fatsemi c_2, v, cc\rangle & \mapsto & \langle c_1, v, \mathit{Fst}\ cc\ c_2\rangle & \mathit{rule}\ 2 \\
\{c_1, v, \mathit{Fst}\ cc\ c_2\} & \mapsto & \langle c_2, v, \mathit{Snd}\ c_1\ cc\rangle & \mathit{rule}\ 3 \\
\{c_2, v, \mathit{Snd}\ c_1\ cc\} & \mapsto & \{c_1 \fatsemi c_2, v, cc\} & \mathit{rule}\ 4 \\
\langle c_1 + c_2, L\ v, cc\rangle & \mapsto & \langle c_1, v, \mathit{LeftPlus}\ cc\ c_2\rangle & \mathit{rule}\ 5 \\
\{c_1, v, \mathit{LeftPlus}\ cc\ c_2\} & \mapsto & \{c_1 + c_2, L\ v, cc\} & \mathit{rule}\ 6 \\
\langle c_1 + c_2, R\ v, cc\rangle & \mapsto & \langle c_2, v, \mathit{RightPlus}\ c_1\ cc\rangle & \mathit{rule}\ 7 \\
\{c_2, v, \mathit{RightPlus}\ c_1\ cc\} & \mapsto & \{c_1 + c_2, R\ v, cc\} & \mathit{rule}\ 8 \\
\langle c_1 \times c_2, (v_1, v_2), cc\rangle & \mapsto & \langle c_1, v_1, \mathit{LeftTimes}\ cc\ c_2\ v_2\rangle & \mathit{rule}\ 9 \\
\{c_1, v_1, \mathit{LeftTimes}\ cc\ c_2\ v_2\} & \mapsto & \langle c_2, v_2, \mathit{RightTimes}\ c_1\ v_1\ cc\rangle & \mathit{rule}\ 10 \\
\{c_2, v_2, \mathit{RightTimes}\ c_1\ v_1\ cc\} & \mapsto & \{c_1 \times c_2, (v_1, v_2), cc\} & \mathit{rule}\ 11 \\
\langle \mathit{trace}\ c, v, cc\rangle & \mapsto & \langle c, R\ v, \mathit{Trace}\ cc\rangle & \mathit{rule}\ 12 \\
\{c, L\ v, \mathit{Trace}\ cc\} & \mapsto & \langle c, L\ v, \mathit{Trace}\ cc\rangle & \mathit{rule}\ 13 \\
\{c, R\ v, \mathit{Trace}\ cc\} & \mapsto & \{\mathit{trace}\ c, R\ v, cc\} & \mathit{rule}\ 14
\end{array}
$$



The derivation of the Π^o interpreter, while tedious, is entirely straightforward. A couple of points however need clarification:

1. The definition here shows only the handling of the composition combinators, which are the interesting cases. All the primitive isomorphisms are hidden

in the $iso(v)$ application in *rule* 1. This should be read as "transform v according to the primitive isomorphism iso."
2. Stuck states in this machine correspond to runtime values not matching their expected types. They can be handled as described in Sec. 3.3 using GADTs (which eliminates them entirely), divergence, or adding extra halting states. We have abstracted from this choice and marked the relevant cases with combinators labeled *cast*.

5 Conclusion

We have developed a technique for the systematic derivation of Π^o programs from logically reversible small-step abstract machines. Since techniques for devising small-step interpreters are well known, this allows for the direct development of a large class of Π^o programs. We have demonstrated the effectiveness of this approach by deriving a meta-circular interpreter for Π^o.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. 1116725.

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/ETAPS. pp. 42–56. Springer-Verlag (2011)
3. Bowman, W.J., James, R.P., Sabry, A.: Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In: Reversible Computation (2011)
4. Cheney, J., Hinze, R.: First-class phantom types. Tech. rep., Cornell Univ. (2003)
5. Danvy, O.: From reduction-based to reduction-free normalization. In: AFP. pp. 66–164. Springer-Verlag (2009)
6. Huelsbergen, L.: A logically reversible evaluator for the call-by-name lambda calculus. InterJournal Complex Systems 46 (1996)
7. James, R.P., Sabry, A.: Information effects. In: POPL. pp. 73–84. ACM (2012)
8. Kluge, W.E.: A reversible SE(M)CD machine. In: International Workshop on Implementation of Functional Languages. pp. 95–113. Springer-Verlag (2000)
9. Rendel, T., Ostermann, K.: Invertible syntax descriptions: unifying parsing and pretty printing. In: Symposium on Haskell. pp. 1–12. ACM (2010)
10. Selinger, P.: A survey of graphical languages for monoidal categories. In: New Structures for Physics, pp. 289–355. Lecture Notes in Physics, Springer (2011)
11. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: POPL. pp. 224–235. ACM (2003)
12. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Conference on Computing Frontiers. pp. 43–54. ACM (2008)
13. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: PEPM. pp. 144–153. ACM (2007)