Theseus:
A High Level Language for Reversible Computing

Roshan P. James and Amr Sabry

Jane Street Capital, New York   Indiana University, Bloomington

Abstract. Programming in a reversible language remains "different" than programming in conventional irreversible languages, requiring specialized abstractions and unique modes of thinking. We present a high level language for reversible programming, called Theseus, that meshes naturally with conventional programming language abstractions. Theseus has the look and feel of a conventional functional language while maintaining a close correspondence with the low-level family of languages Π based on type isomorphisms [9]. In contrast to the point-free combinators of Π, Theseus has variables and binding forms, algebraic data types, function definitions by pattern matching, and is Turing complete. The language is strongly typed and all well-typed programs are reversible. We explain the semantics of Theseus via a collection of progressively expressive examples and outline its correspondence to Π.

Keywords: Type Isomorphisms; Information Effects; Monoidal Categories

1 Introduction

The main contribution of this paper is a new language design for reversible computing called *Theseus*. Theseus is a high-level language inspired by our previously-developed information-preserving family of languages called Π (with various subscripts) [2, 10, 9, 8]. This family of languages is itself inspired by the physical principle of *conservation of information* which was formally captured by requiring every computation in a Π language to be a combinator witnessing a *type isomorphism*. Programming with combinators, i.e., in a point-free style with no variables, is generally an awkward task that is compounded by the reversibility invariant. Furthermore, naïvely introducing variables immediately destroys the tight control over information flow required for reversibility. Furthermore, our experience with conventional linear type disciplines showed that these are not sufficient to track information flow across *choices* (i.e., across conditional expressions) [16]. Our solution, exhibited in the design of Theseus, is to restrict variable introduction to pattern matching clauses and to augment conventional pattern matching rules with a few additional restrictions that guarantee information preservation with a familiar look and feel for programmers. [1]

Since the design of reversible programming languages is now a relatively mature subject, we start with a short overview of approaches to reversible computing to provide a wider context for Theseus. There are a few major themes that dominate the design

[1] An implementation of Theseus is available from: `https://bitbucket.org/roshanjames/theseus`.

of reversible computational models. Several approaches to reversible computation fall under the broad category of *circuit models*. The very first reversible models to be studied widely led to the design of the Toffoli gate [17], the Fredkin gate, and what was dubbed "*conservative logic*" [4]. More recent approaches include Morita's Rotary elements [14] and circuit models born from the study of the *Geometry of Interaction* [12]. The work by Abramsky, Coecke and others introduced a class of circuit models based on categorical string diagrams as a mathematical basis for quantum physics and quantum computation [1]. The common theme among the circuit models is that the setting in which computation takes place is that of a circuit wherein the process of computation either traces the flow of values in the circuit or is expressed as rewrites of circuits until some normal form is achieved. Programming with circuit models feels very different from conventional programming because most circuit models deal directly with a graphical representation and few have term languages. Quipper [6] is an embedded domain-specific language in Haskell aimed as making it easy to specify quantum circuits. In its current design however, Quipper lacks a linear type system which means that it lacks static guarantees of reversibility. Another class of reversible languages come from the notion of program inversion. These include compiling source programs adding enough state (possibly including execution traces) such the resulting program's execution can be inverted [3, 7, 15, 13].

Janus. Probably the most well-developed high-level language for reversible computing is Janus, which was originally designed by Lutz and Derby in 1982 [11] and developed significantly in recent years by Yokoyama and Glück [20, 5]. Janus is an *imperative* language wherein every primitive statement is reversible. Procedures can be *called* or *uncalled*, corresponding to executing them forwards or backwards. Janus allows for integers and array values, as well as stacks for dynamic allocation of memory. Values are passed by reference to functions and the same value may not be referenced twice to avoid aliasing issues. Integers have fixed precision and overflows/underflows allow for typical arithmetic operations to always be well-defined.

The main difficulty with writing large Janus programs arises from the treatment of local variables and control flow. Local variables can be allocated by specifying their initial values and *deallocated* by specifying the value they must hold at deallocation time. Further, Janus contains a structured control flow mechanism where predicates are attached to join points of the program. These predicates must be chosen such that the backwards execution flow of the program is deterministic and chooses the execution path that traces back the path that forward execution would have taken. While Janus maintains the look and feel of a conventional imperative language for forward execution, it requires considerable effort from the programmer to get reverse execution right. Both the choice of deallocation-time values and predicates for inverse control flow are application specific choices that the programmer must determine on a case-by-case basis. These are hard to determine in general and can result in programs that will not correctly execute in reverse.

Theseus. Theseus makes a fundamentally different choice from Janus. In Theseus all well-typed programs are reversible and programmers can reason about the reversibility of the programs by following straightforward syntactic and type correctness requirements. We have chosen to name this high-level language Theseus in the spirit of the paradox of equality called the *Ship of Theseus* and, like in the Greek legend, computation in Theseus proceeds by replacing values by apparently equal values. The design of Theseus has similar high-level goals to other *functional* reversible languages such as the language proposed by Yokoyama, Axelsen, and Glück [19] but differs significantly in the details. In Theseus all functions are partial isomorphisms and patterns cannot overlap which makes every computational step evidently and directly reversible. In the proposed language of Yokoyama et. al, functions are injective, duplication is allowed in controlled ways, and patterns may overlap. These features make the language closer to conventional irreversible languages, and hence amenable to more familiar programming techniques but at the cost of complicating the language.

Structure. The remainder of the paper is structured as follows. We first review the language Π^o which Theseus generalizes and compiles to. In Sec. 3, we introduce the basic ingredients of Theseus relating to types and pattern matching which allow the definitions of many standalone reversible functions. Sec. 4 extends the basic definitions with the ability to parametrize functions by other functions giving the programmer the expressiveness to compose computations. The next section extends the language further with "iteration labels" which allows recursive definitions to be written in a direct style. Sec. 6 concludes and puts our work in perspective.

2 A Quick Review of Π^o

Theseus aims to be a high level language for Π, Π^o, and their variants. For comparison purposes, we briefly review the reversible language Π^o in this section. As we illustrate, programming in Π^o requires considerable effort in tracking and manipulating values in a point-free style as the main programming abstraction is that of a "diagram."

The set of types includes the empty type 0, the unit type 1, sum types $b_1 + b_2$, product types $b_1 * b_2$, 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 a product type, and $\langle v \rangle$ which builds recursive values. There are no values of type 0. The expressions of Π^o are witnesses of type isomorphisms:

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

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

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 $(b, +, 0, *, 1)$

is a *commutative semiring*, i.e., that each of $(b, +, 0)$ and $(b, *, 1)$ is a commutative monoid and that multiplication distributes over addition. The last isomorphism witnesses the equivalence of a value of a 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:

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

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

Following the tradition for computations in monoidal categories, Π^o has a graphical notation that conveys its semantics. 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 as detailed in previous work [10, 8]. Symbolically, a complete program consists of a circuit c and a value v and the process of evaluation $c(v)$ consists of the flow of the value v through the circuit.

Definition 1 (Logical reversibility [21]). *A map $c_1 : b_1 \to b_2$ is logically reversible if there exists an inverse map $c_2 : b_2 \to b_1$ such that for all values $v_1 : b_1$ and $v_2 : b_2$, we have: $c_1(v_1) = v_2$ iff $c_2(v_2) = v_1$.*

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., $(c_1 \fatsemi c_2)^\dagger = (c_2{}^\dagger) \fatsemi (c_1{}^\dagger)$. All Π^o computations are *logically reversible* [9] where the inverse of c is given by $c^\dagger$.

3 Theseus: Types and Simple Isomorphisms

We begin by presenting the core of Theseus which consists of evidently reversible functions on user-defined datatypes. In subsequent discourse we will use the word 'map' to refer to these logically reversible functions. Theseus has the same built-in types as Π^o and as conventional functional programming languages: the empty type, the unit type, and sum and product types. All other types, including recursive types, are user-defined using a general mechanism for type declarations:

```
type Bool  =  True | False
type Nat   =  0 | Succ Nat
type Tree  =  Leaf Nat | Node Tree Tree
```

The declarations above witness isomorphisms between the set `Bool` and the set constructed by the constants `True` and `False`, between the set `Nat` and the set inductively constructed using `0` and `Succ`, and between the set `Tree` and the set inductively constructed using `Leaf` and `Node`. Thus, Theseus type definitions are similar to, say, Haskell, type definitions. Such definitions translate directly to the underlying Π^o using iso-recursive type definitions. For example, the above types translate as follows:

$$\begin{array}{lrcl}
\texttt{Bool} = \mu\, x.1 + 1 & \textit{unfoldBool} : & \texttt{Bool} \leftrightarrow 1 + 1 & : \textit{foldBool} \\
\texttt{Nat} = \mu\, x.1 + x & \textit{unfoldNat} : & \texttt{Nat} \leftrightarrow 1 + \texttt{Nat} & : \textit{foldNat} \\
\texttt{Tree} = \mu\, x.\texttt{Nat} + x * x & \textit{unfoldTree} : & \texttt{Tree} \leftrightarrow \texttt{Nat} + (\texttt{Tree} * \texttt{Tree}) & : \textit{foldTree}
\end{array}$$

3.1 Simple Isomorphisms

A Theseus program has a type of the form `a ↔ b`; one runs such a program forward by supplying a value of type `a`. If the program terminates, we get back a value of type `b`. Alternately we can run the program in reverse by supplying a value of type `b`. Given the built-in types and the facility to introduce user-defined ones, one can already write a few small but interesting programs. Maps that are evidently isomorphisms can be written in the familiar pattern matching style popularized by the family of functional languages. For example:

```
id :: Bool ↔ Bool
| False  ↔  False
| True   ↔  True

expandBool :: Bool * a ↔ a + a
| True, a    ↔  Left a
| False, a   ↔  Right a

expandNat :: Nat ↔ Nat + 1
| 0        ↔  Right ()
| Succ n   ↔  Left n
```

```
not :: Bool ↔ Bool
| True   ↔  False
| False  ↔  True

foldBool :: a + a ↔ Bool * a
| Left a    ↔  True, a
| Right a   ↔  False, a

foldNat :: Nat + 1 ↔ Nat
| Right ()   ↔  0
| Left n     ↔  Succ n
```

```
treeUnwind :: Tree ↔ Tree * Tree + (Bool + Nat)
| Node t1 t2              ↔  Left   (t1, t2)
| Leaf 0                  ↔  Right  (Left True)
| Leaf (Succ 0)           ↔  Right  (Left False)
| Leaf (Succ (Succ n))    ↔  Right  (Right n)
```

Unlike the situation in a conventional functional language, the intuition here is that the two sides of each pattern clause can be swapped to produce the inverse of the function, i.e. as we will see repeatedly in the paper, patterns and expressions are the same thing. Compare for example the definitions of `expandBool` and `foldBool`. This is the only constraint that a programmer needs to maintain. This constraint can be ensured using the following two rules:

1. *Non-overlapping and exhaustive coverage in pattern clauses.* The collections of patterns in the left-hand side (LHS) of each clause must be a complete non-overlapping covering of the input type. Similarly, the collections of patterns in the right-hand side (RHS) of each clause must also be a complete non-overlapping covering of the return type. In other words, no cases should be omitted or duplicated.
2. *Preserve typed variables across* ↔. Each variable that occurs on one side of a pattern matching clause can only appear once on that side and must appear exactly once on the other side and with the same type.

When restricted to one side of each pattern matching clause, the rules should be familiar and intuitive. Note that, in general, the two sides of a pattern matching clause may have different types. Other than the requirement of non-overlapping and exhaustive coverage, there are no rules that constrain the types and names of constants nor of constructors. So for example, in `foldBool`, the pattern '`Left a`' maps to a pair constructor with the first component being the constant `True`.

Examples of expressions that violate the constraints follow:

```
-- Invalid: Missing LHS cases
missing_node :: Tree ↔ Nat
| Leaf n ↔ n

-- Invalid: Overlapping RHS cases
overlapping_cases :: Nat ↔ Nat
| 0       ↔  0
| Succ n  ↔  n
```

```
-- Invalid: n is dropped
drop_var :: Tree ↔ Tree * Tree + Nat
| Node t1 t2  ↔  Left  (t1,t2)
| Leaf n      ↔  Right 0

-- Invalid: t is used twice
dup_var :: Tree ↔ Tree * Tree + Nat
| Node t t  ↔  Left  (t,t)
| Leaf n    ↔  Right n
```

It is easy to see that every primitive isomorphism of Π^o can be expressed in Theseus. Here are a few examples.

```
assocL :: a+(b+c) ↔ (a+b)+c :: assocR
| Left a            ↔ Left (Left a)
| Right (Left b)    ↔ Left (Right b)
| Right (Right c)   ↔ Right c
```

```
zeroe :: b + 0 ↔ b :: zeroi
| Left v ↔ v

swapTimes :: a * b ↔ b * a
| (a,b) ↔ (b,a)
```

Every simple isomorphism of Theseus can be translated to Π^o as follows:

1. For every Theseus map `c : a ↔ b`, we can define an intermediate type `ti` which is the sum of pairs of all the variables that occur in the LHS patterns. Since the LHS and RHS patterns contain the same type variables, `ti` is unique for a given `c` (up to ordering). For example, in `treeUnwind` the first clause has variables `t1 : Tree` and `t2 : Tree`, the second and third clauses have no variables and the fourth clause has the variable `n : Nat`. The expected type `ti` is therefore `Tree * Tree + (1 + (1 + Nat))`, where we insert the type `1` as a placeholder for the clauses that did not have type variables.
2. By construction `ti` is isomorphic to `a` and `b`. Hence, maps `c1 : a ↔ ti` and `c2 : ti ↔ b` must be expressible in Π^o. (Both the above isomorphisms can be constructed by systematically unfolding and distributing `a` and `b` until `ti` is obtained. The exact details of such an expansion can be found in Sec. 3 of our previous paper [10] and are skipped in this paper.) Once values of type `a` and `b` can be mapped to values of type `ti`, the required translation of `c` is given by `c1 ⨾ c2`.

3.2 Dealing with Numbers

If we try to do arithmetic using the previously defined Peano-style `Nat`, we have to address an issue of what to do when we encounter `sub1 0`. There are two obvious choices:

1. We can define `add1` and `sub1` of type `Nat ↔ Nat`, such that `sub1` of `0` diverges. We show how these maps can be defined in Sec. 5.
2. An error mechanism can be added to the language such that when an error is raised program execution is undefined and there is no meaningful inverse definition.

Another choice, similar to that made by Janus, is to use bounded integers such that every operation is always well defined through underflows and overflows. Here is a simple 4-bit `Nat4` datatype with its corresponding `add1` and `sub1` operations:

```
type Nat4 = Bool * Bool * Bool * Bool

add1 :: Nat4 ↔ Nat4 :: sub1
| (a, b, c, False)          ↔  (a, b, c, True)
| (a, b, False, True)       ↔  (a, b, True, False)
| (a, False, True, True)    ↔  (a, True, False, False)
| (False, True, True, True) ↔  (True, False, False, False)
| (True, True, True, True)  ↔  (False, False, False, False)
```

It is easy to see how a math library may be defined in this way. Once an operator can be defined in Theseus it is always possible to replace its implementation by an equivalent one that is expressible efficiently in the underlying hardware. In this case one could compile down to the CPU's native integer representation and operators.

4 Parametrized Maps

Now that we can express simple standalone isomorphisms, we explain how to compose such isomorphisms to model more complex behavior. In Π^o, there are three ways of composing isomorphisms: sequential composition, parallel composition, and choice composition. The common idiom underlying these composition combinators is that a reversible map can be parametrized by another reversible map. This idea is related to "higher-order functions" but is more limited as we explain below.

4.1 Definition and Examples

A Theseus map `f` can be parametrized by another map `g` by adding a labeled argument `g` of the appropriate type to `f`. In the example below `treeUnwindf` is parametrized over some map `f` which it applies to `n`, if the supplied tree is `Leaf n`:

```
treeUnwindf :: f:(Nat ↔ a) → Tree ↔ Tree * Tree + a
| Node t1 t2 ↔ Left (t1, t2)
| Leaf n     ↔ Right (f n)
```

This parametrization should be thought of as a macro or a meta-language construction. Theseus does not have high-order maps in the formal sense. In other words, the final type of a Theseus program must be of the form `a` ↔ `b` and every occurrence of an arrow type → must be instantiated at compile time. The notation for instantiating the label parameter `f` in the map `fun` by the map `g` is `fun ~f:g`. For example, `treeUnwindf ~f:expandNat` should be thought of as shorthand for the map:

```
_ :: Tree ↔ Tree * Tree + (1 + Nat)
| Node t1 t2    ↔ Left (t1, t2)
| Leaf 0        ↔ Right (Right ())
| Leaf (Succ n) ↔ Right (Left n)
```

which inlines `expandNat` within the definition of `treeUnwindf`.

We discuss the reversibility of parametrized maps in the next section. We first point out that the Π^o closure primitives can be expressed as parametrized maps:

```
_._ :: f:(a ↔ b) → g:(b ↔ c) → a ↔ c
| a ↔ g (f a)

_*_ :: f:(a ↔ b) → g:(c ↔ d) → a * c ↔ b * d
| (a,c) ↔ (f a , g c)

_+_ :: f:(a ↔ b) → g:(c ↔ d) → a + c ↔ b + d
| Left a   ↔  Left (f a)
| Right c  ↔  Right (g c)
```

As an additional example, parametrized maps allow us to define controlled operations in the tradition of reversible circuits. In the definition below, the boolean input is a control bit that determines which of the maps `th` or `el` is applied to the second component of the pair. This conditional is then used to define the "controlled-not" gate:

```
if :: th:(a ↔ b) → el:(a ↔ b) → Bool * a ↔ Bool * b
| True, a  ↔ Left (th a)
| False, a ↔ Right (el a)

cnot :: Bool * Bool ↔ Bool * Bool
| control, bit ↔ if ~th:not ~el:id (control, bit)
```

4.2 Reversing Parametrized Maps

Reversing a map of type `a` ↔ `b` results in a map of type `b` ↔ `a` which is its adjoint. As discussed before, simple maps can be reversed by simply switching the LHS and RHS for both the type and the pattern matching clauses comprising the body. Interestingly, one can think about the reverse of parametrized maps in the same way – i.e. one simply needs to switch LHS and RHS of the '↔'.

```
treeUnfold :: f:(Nat ↔ a) → Tree ↔ Tree * Tree + a
| Node t1 t2  ↔ Left (t1, t2)
| Leaf n      ↔ Right (f n)

-- the adjoint of treeUnfold
rev_treeUnfold :: f:(Nat ↔ a) → Tree * Tree + a ↔ Tree
| Left (t1, t2)  ↔ Node t1 t2
| Right (f n)    ↔ Leaf n
```

Note that the application `f n` happens on the left of the ↔, which is something that one does not see in conventional functional languages. These applications in the LHS have the dual meaning of applications in the RHS and should be understood as follows: the value that is the actual argument of the `Right` constructor is the result of applying `f n`. In the forward execution of `rev_treeUnfold`, when the actual value of the constructor argument is encountered, it is passed through the reverse of `f` to get the value `n`. In other words, the above program is exactly the same as `rev_treeUnfold'` below where `rev_f` is the adjoint of `f`:

```
rev_treeUnfold' :: rev_f:(a ↔ Nat) → Tree ↔ Tree * Tree + a
| Left (t1, t2) ↔ Node t1 t2
| Right a       ↔ Leaf (rev_f a)
```

It follows that the following programs are equivalent:

```
g :: f:(a ↔ b) → a ↔ b                   g :: rev_f:(b ↔ a) → a ↔ b
| a ↔ f a                                | rev_f b ↔ b
```

Based on the above observation, it is easy to define the `sym` map that constructs the adjoint of any given map:

```
sym :: f:(a ↔ b) → b ↔ a
| (f a) ↔ a
```

While parametrized maps add tremendous programming convenience to Theseus, they don't change the expressive power of the language. All programs expressible with parametrized maps, can be expressed without them by fully inlining the actual parameters. Thus compiling Theseus programs with parametrized maps to Π^o simply requires first inlining the maps and then doing the translation described in Sec. 3.1.

5 Iteration Labels

The fragment of Theseus discussed up to now, modulo full recursive types, can be expressed in pure Π without *trace*. Introducing recursive definitions in a reversible language is subtle because every iteration must be reversible which requires, at least, that the number of iterations be recoverable from the output. The insight we use in Theseus is that this can be achieved elegantly by adding *typed labels* as explained below. We start with a small example:

```
parity :: Nat * Bool ↔ Nat * Bool
| n, b                ↔ iter $ n, 0, b
| iter $ Succ n, m, b ↔ iter $ n, Succ m, not b
| iter $ 0, m, b      ↔ m, b
  where iter :: Nat * Nat * Bool
```

The map `parity` calculates whether the given natural number is even or odd by counting to `0` and flipping the second boolean input each time. In the definition above, the label appears twice on the LHS and twice on the RHS. The correctness (i.e., reversibility) of the map is guaranteed as labels obey the same restrictions as the one for simple patterns. In particular the occurrences of the label in the LHS must constitute a non-overlapping coverage of the label type, and similarly for the occurrences of the label in the RHS.

The important thing to note about labels is that they may have a different type from the LHS or RHS types of their containing map. In this case for example, the `parity` map has **`Nat * Bool`** as its LHS and RHS type while the type of the label **`iter`** has the type **`Nat * Nat * Bool`**. Labels temporarily give us a different *view* [18] of a value, by changing it type. Labels act as goto statements from one side to the other – when a label is encountered on the RHS of a map, the argument to the label is matched by some definition of the label pattern on the LHS of the map, and vice versa. Here is a trace of the execution of `parity` as it transforms input `Succ (Succ (Succ 0)), False` to the output `Succ (Succ (Succ 0)), True`. It takes five pattern match and rewrite steps to complete.

```
Succ (Succ (Succ 0)), False           ⟼ iter $ Succ (Succ (Succ 0)), 0, False
iter $ Succ (Succ (Succ 0)), 0, False ⟼ iter $ Succ (Succ 0), Succ 0, True
iter $ Succ (Succ 0), Succ 0, True    ⟼ iter $ Succ 0, Succ (Succ 0), False
iter $ Succ 0, Succ (Succ 0), False   ⟼ iter $ 0, Succ (Succ (Succ 0)), True
iter $ 0, Succ (Succ (Succ 0)), True  ⟼ Succ (Succ (Succ 0)), True
```

A trace operator can be expressed using labels and the definition closely follows the expected iteration semantics of *trace* in our previous work [9]. Using `trace` one can define `add1` on `Nat` such that `sub1` of `0` diverges.

```
trace :: f:(a + b ↔ a + c) → b ↔ c
| b               ↔ enter $ Right b
| enter $ a       ↔ leave $ (f a)
| leave $ Left a  ↔ enter $ Left a
| leave $ Right c ↔ c
  where enter :: a + b
        leave :: a + c
```

```
addSub :: Nat + Nat ↔ Nat + Nat
| Left (Succ n) ↔ Left n
| Left 0        ↔ Right 0
| Right n       ↔ Right (Succ n)

add1 :: Nat ↔ Nat :: sub1
| n ↔ trace ~f:addSub n
```

Here are a few more examples that use labels. Combinators `iterN` and `add` are ones you would expect to find in a standard library. The combinator `iterN` loops any given map `n` times and `add` adds the second argument to the first:

```
-- runs f some n-times
iterN :: f:(a↔a) → Nat*a ↔ Nat*a
| n, a                ↔ iter $ n, 0, a
| iter $ 0, n, a      ↔ n, a
| iter $ Succ n, m, a ↔
    iter $ n, Succ m, f a
 where iter :: Nat * Nat * a
```

```
-- add (x, y) = (x+y, x)
add : Nat * Nat ↔ Nat * Nat
| x, y ↔ iter $ y, 0, x
| iter $ a, b, 0      ↔ a, b
| iter $ a, b, Succ n  ↔
      iter $ add1 a, add1 b, n
  where iter :: Nat * Nat * Nat
```

As another example, here is definition of the Fibonacci function where `fibonacci` applied to `(n, (1, 0))` returns `(n, (x, y))` where `x` is the $(n + 1)$-st Fibonacci number and `y` is the previous one:

```
swapAndAdd :: Nat * Nat ↔ Nat * Nat
| x, y ↔ add (y, x)

fibonacci :: Nat * (Nat * Nat) ↔ Nat * (Nat * Nat)
| n, (x, y) ↔ iterN ~f:swapAndAdd (n, (x, y))
```

We conclude this section with a somewhat richer example of tree traversal that shows how we can work with general recursive types. Here we use two labels **`walk`** and **`reconst`** and one can verify that each label independently covers its type on both the LHS and the RHS. The combinator `treeWalk` walks down a tree and applies a given `f` to every leaf:

```
type Ctxt = Empty | L Ctxt Tree | R Tree Ctxt

treeWalk :: ~f:(Nat ↔ Nat) → Tree ↔ Tree
| tr                      ↔ walk $ tr, Empty
| walk $ Leaf b, ctxt     ↔ reconst $ ctxt, Leaf (f b)
| walk $ Node b1 b2, ctxt ↔ walk $ b1, L ctxt b2
| reconst $ Empty, tr     ↔ tr
| reconst $ L ctxt b2, b1 ↔ walk $ b2, R b1 ctxt
| reconst $ R b1 ctxt, b2 ↔ reconst $ ctxt, Node b1 b2
  where walk    :: Tree * Ctxt
        reconst :: Ctxt * Tree
```

Compiling programs with labels to Π^o is a straightforward extension of compiling label-free programs. For any Theseus map `c : a` ↔ `b` that uses an iteration label **`label`** `: lab` we construct an intermediate type `ti` corresponding to the pattern clauses as before. We can then construct the isomorphisms `c1 : lab + a` ↔ `ti` and `c2 : ti` ↔ `lab + b` since every LHS pattern clause corresponds to either `a` (the input) or `lab` (the label) and every RHS clause corresponds to either `b` (the output) or `lab` (the label). The resulting `c1` $\fatsemi$ `c2` has type `lab+a` $\leftrightarrow$ `lab+b` and the required compilation of `c` is given by *trace*(`c1` $\fatsemi$ `c2`) `:` `a` $\leftrightarrow$ `b`. This extends to multiple by labels by constructing `c1 : (lab1 + lab2 + ...) + a` $\leftrightarrow$ `ti` and `c2 : ti` $\leftrightarrow$ `(lab1 + lab2 + ...) + b` to include the types of the additional labels.

6 Conclusion

Theseus is as expressible as Π^o and preserves its properties. All Π^o programs are expressible as Theseus programs and vice-versa. In other words, Π^o, or dagger symmetric traced bimonoidal categories serve as a fully abstract model for Theseus. Despite being closely related to Π^o, programmers can use Theseus without needing to understand Π^o, strings diagrams or category theory.

Theseus admits non-terminating computations. Just the way that partial functions refer to maps that may not be defined on all inputs, the set of computations expressible in Theseus are referred to as *partial isomorphisms*. As with Π^o, Theseus programs satisfy *logical reversibility* (see Def. 1).

Many of the trimmings of more mainstream languages can be added to Theseus. Theseus could use an error reporting mechanism for programs to exit to without resorting to divergence. Theseus needs an FFI mechanism to take advantage of third-party libraries. We imagine that in such cases the programmer would write out an explicit extern declaration asserting the type that Theseus should assume the external library has. More generally, Theseus can adopt Agda's approach of piggy backing on top of Haskell. We can have an FFI that lets us drop into Haskell and use Haskell API. We could also import user-certified pairs of Haskell functions as isomorphisms into Theseus.

There are few natural extensions to Theseus that require additional research, though preliminary investigation seems promising. Theseus currently treats all values the same and does not classify values as heap and garbage. Theseus can be extended with a framework for encapsulating effects. In other words, add the effectful arrow type $\rightsquigarrow$ in a principled way such that information effects, and possibly other effects, may be expressed [9]. It is also desirable to add other programming features such as polymorphism, higher-order maps and inductive definitions over recursive datatypes without explicitly managing `Ctxt` values as we did in the `treeWalk` program.

There are many ways to look at what has been achieved here. When working with categories, one typically has a few standard representation tools at their disposal. First, one can write out objects and composition in a category and reason algebraically. Second, one can draw commuting diagrams and reason about these diagrams. Third, one can draw string diagrams and reason about them either by tracing the flow of values or by graph rewriting. Here we suggest another approach: design a programming language corresponding to the category and reason about programs in the language as a means of reasoning about constructions in the category. With Theseus, we answer the question 'What is a programming language corresponding to a category?', rather than the opposite question which is asked more commonly. Namely, 'What, if any, is the categorical model underlying a programming language?'. Even though Theseus presents one such instance, we suspect the approach can be extended to other monoidal categories.

Acknowledgement.

This material is based upon work supported by the National Science Foundation under Grant No. 1217454.

References

1. S. Abramsky and Bob Coecke. A Categorical Semantics of Quantum Protocols. In *LICS*, 2004.
2. William J. Bowman, Roshan P. James, and Amr Sabry. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *Reversible Computation*, 2011.
3. Michael P. Frank. *Reversibility for efficient computing*. PhD thesis, Massachusetts Institute of Technology, 1999.
4. E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.
5. Robert Glück and Masahiko Kawabe. Revisiting an automatic program inverter for Lisp. *SIGPLAN Not.*, 40:8–17, May 2005.
6. Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A Scalable Quantum Programming Language. *SIGPLAN Not.*, pages 333–342, 2013.
7. David Gries. *The Science of Programming. ch:21 Inverting Programs*, volume 1981. Springer Heidelberg, 1981.
8. Roshan P. James. *The Computational Content of Isomorphisms*. PhD thesis, Indiana University, Bloomington, 2013.
9. Roshan P. James and Amr Sabry. Information effects. In *POPL*, pages 73–84. ACM, 2012.
10. Roshan P. James and Amr Sabry. Isomorphic interpreters from logically reversible abstract machines. In *Reversible Computation*, 2012.
11. Christopher Lutz. Janus: a time-reversible language. *Letter to R. Landauer*. `http://www.tetsuo.jp/ref/janus.html`, 1986.
12. Ian Mackie. Reversible higher-order computations. In *Reversible Computation*, 2011.
13. Armando B. Matos. Linear programs in a simple reversible language. *Theor. Comput. Sci.*, 290:2063–2074, January 2003.
14. Kenichi Morita. A simple universal logic element and cellular automata for reversible computing. In *Machines, Computations, and Universality*, Lecture Notes in Computer Science, pages 102–113. Springer Berlin Heidelberg, 2001.
15. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An Injective Language for Reversible Computation. In *Mathematics of Program Construction*, 2004.
16. Zachary Sparks and Amr Sabry. Superstructural reversible logic. In *International Workshop on Linearity*, 2014.
17. Tommaso Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.
18. P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM.
19. Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a reversible functional language. In *Reversible Computation*, 2011.
20. Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *PEPM*, pages 144–153. ACM, 2007.
21. P. Zuliani. Logical reversibility. *IBM J. Res. Dev.*, 45:807–818, November 2001.