Reversible Computing: Theseus, Matrices, & Circuits

Final Project for B629

BY CHENCHAO DING

cd17@iu.edu

Project Description

The relations and correspondences among the high-level reversible proramming languages, matrices and circuits are intriguing. They are logical, algebraic and physical respectively, while they intuitively share similar expressive power in terms of reversible computing. Compilation from a subset of Theseus to the reversible circuits will be briefly covered in this project. Some other topics like abstract algebra, propositional logic and programming languages will be slightly mentioned as examples and extensions. The aim of this project is to provide a motivation and explore the relations among reversible programming languages, matrices and circuits.

1 Motivation: A Tiny Example

Quantum programming languages like Qiskit and QASM have already demonstrated how reversible computing works from the view of reversible circuits. However, such languages are no more than a straightforward description of low-level circuits: the computation paradigm is way too different from traditional high-level programming languages. What is the gap between? Where does the reversibility come from? How can a program evolve to its "reversible version"?

To fill the gap, we first go back to the very fundamental of programming languages.

The traditional irreversible computing is in general the process of recursive evaluation of expressions to the value (reduction to the normal form) which "loses information" about intermediate, temporary values. Here is a tiny piece of code written in Racket:

```
Scheme] (let ((a 1)
              (b 1)
              (c 1)
              (d 1))
          (and (or a b)
               (or c d)))
```

```
1
```

The nested expression `(and (or a b) (or c d))` can be viewd as a function with 4 arguments. By setting all of them to be 1, it evaluates to 1.

```
Scheme] ((lambda (a b c d)
           (and (or a b)
                (or c d)))
         1 1 1 1)
```

```
1
```

We cannot run backwards to obtain the exact input since the result is not unique, eg. `(1 0 1 0)` will also map to `1`, but the most essential reason here is that the inputs and intermediate values are simply thrown away after used. The data flow diagram shows the idea more clearly.

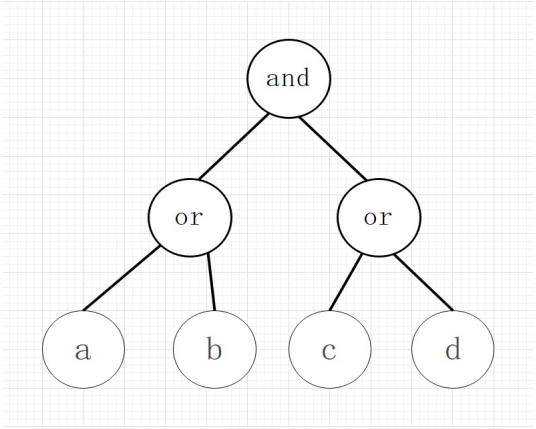

Figure 1. syntax tree for `(and (or a b) (or c d))`

Unnesting the syntax tree (see **Figure 1**) will expose the data flow (see **Figure 2**), and temporary "points" through it.

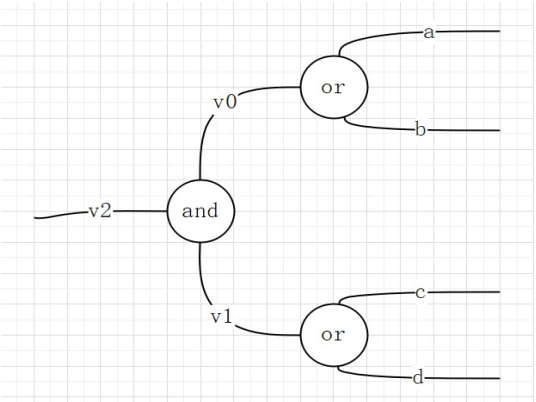

Figure 2. data flow for `(and (or a b) (or c d))`

Below is a Racket program that unnests, or flattens the syntax tree and exposes intermediate variables in the data flow (it also forces specification of the evaluation order, but it isn't interesting in this project).

```
Scheme] (define unnest
          (λ (exp C)
            (match exp
              [(? symbol? x) (C x)]
              [`(,op ,e1 ,e2)
               (unnest e1
                       (λ (v1)
                         (unnest e2
                                 (λ (v2)
                                   (let ([v* (gensym 'v)])
                                     (match op
                                       [(or 'or 'and)
                                        `(let ([,v* (,op ,v1 ,v2)])
                                           ,(C v*))])))))))])))
```

```
Scheme] (unnest '(and (or a b) (or c d))
                (lambda (x) x))

   '(let ((v0 (or a b)))
      (let ((v1 (or c d)))
        (let ((v2 (and v0 v1)))
          v2)))
```

Here `v0`, `v1` and `v2` along with 4 arguments are all "points" generated during evaluation: `a`, `b`, `c` and `d` are inputs variable; `v2` is the output variable; `v0` and `v1` are intermediate variables discarded after evaluation (while they are important in reversible evaluation, as shown below).

Essentially, there are no differences between those "points"/intermediate variables and arguments of functions like `a`, `b`, `c`, and `d`: they are both "wires" carrying some values in the data flow diagram. Therefore, in theory we can design a correctness preserved "point-free" program, which consists of only one big function (may be composed by several small functions) applying to a set of "initial points" to get another set of "output points" like how matrices, SKI combinators and composed functions work.

$$\begin{aligned} \boldsymbol{CBA} \cdot \mathbf{v} \\ ((\boldsymbol{SKK})\, x) \\ (f \circ g)[x, y] \end{aligned}$$

Being "point-free" doesn't guarantee reversibility, but it does eliminate the "infomation loss" due to discarded intermidiate variables, since its data flow graph has only "initial points" and "output points" connected by a "point-free pipe" (can be a matrix, an composed function, or a circuit).

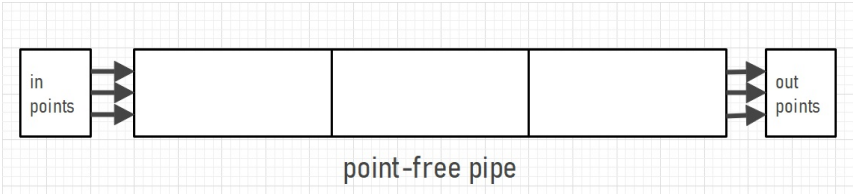

Figure 3. point-free pipe (prototype of reversible circuits)

Is there some "point-free" and even reversible version for `(and (or a b) (or c d))`?

Consider the registerized form of the program:

```
Scheme] (define-register a b c d v0 v1 v2)
        (begin
          (set! v0 (or a b))
          (set! v1 (or c d))
          (set! v2 (and v0 v1)))
```

Here are 7 registers for 7 "points" mentioned above. After every single computation, the register state changes. If we record every register frame (see **Table 1**), it's easy to derive a reverse computation trace: just flip all arrows in the state transfer diagram. Each transition is a reversible function, or matrix. By composing them we have:

$$\begin{aligned} (f_0 \circ f_1 \circ f_2)\boldsymbol{s}_0 &= \boldsymbol{s}_3 \\ (f_2^{-1} \circ f_1^{-1} \circ f_0^{-1})\boldsymbol{s}_3 &= \boldsymbol{s}_0 \\ \boldsymbol{CBA} \cdot \boldsymbol{s}_0 &= \boldsymbol{s}_3 \\ \boldsymbol{A}^{-1}\boldsymbol{B}^{-1}\boldsymbol{C}^{-1}\boldsymbol{s}_3 &= \boldsymbol{s}_0 \end{aligned}$$

Where $\boldsymbol{s}_0$ represents the initial state vector and $\boldsymbol{s}_3$ the output state vector.

a	1		1		1		1
b	1		1		1		1
c	1		1		1		1
d	1	$f_0, \boldsymbol{A}$	1	$f_1, \boldsymbol{B}$	1	$f_2, \boldsymbol{C}$	1
v0	0		1		1		1
v1	0		0		1		1
v2	0		0		0		1

Table 1. register frames throughout evaluation.

Now the question is how can we connect wires (i.e. define and compose those reversible operaters)? We can guess that f_0 and f_1 are something like "reversible OR gate", and f_2 "reversible AND gate".

Back to the nested expreesion. It has two sub-expression `(or a b)`, `(or c d)`. It's natural to keep their evaluation result in order to achieve some kind of reversibility. However, even these two minimal redexes are not reversible since the OR gate used here is not reversible! Now the need for reversible gates is clear.

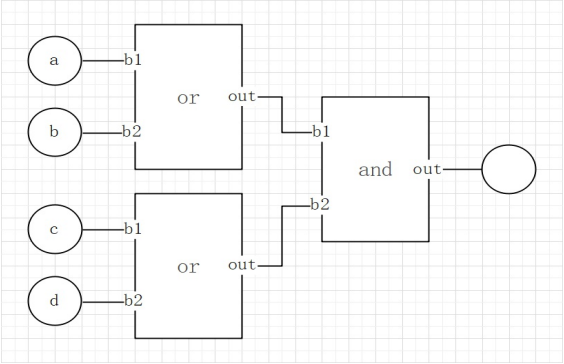

Figure 4. irreversible circuit for `(and (or a b) (or c d))`

Generally, every sub-part of an expression should be reversible individually. Recursively follow the rule "keep evaluation result of sub-expressions", we can build a conceptual reversible version of these logical gates.

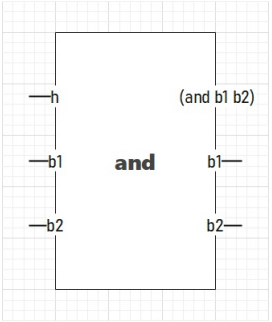

Figure 5. conceptual reversible logical gate

Its generalized form is $U_f(h, b_1, ..., b_n) = (f(b_1, ..., b_n) \oplus h, b_1, ..., b_n,)$, which is frequently seen in Quatum computing literatures.

For each sub-expression with the pattern `(,op ,e1 ,e2)` there should be one bit initialized to carry its evaluation result (record the data flow of intermediate points). In this example, there are 3 matched patterns:

- 2 inner `(or .. ..)`
- 1 outer `(and .. ..)`

Notice that they correspond to `v0~v2` in the data flow diagram (**Figure 2**).

The final imaginary reversible circuit looks like:

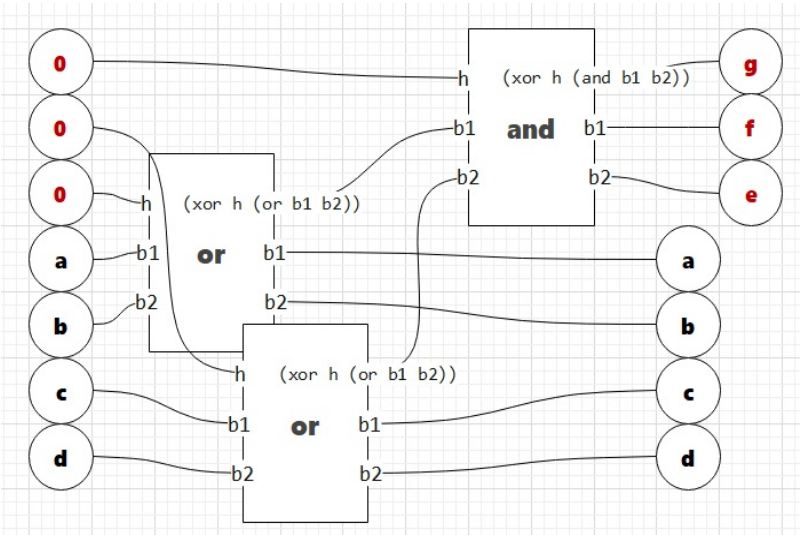

Figure 6. conceptual reversible circuit for `(and (or a b) (or c d))`

The wires are connected. It's very close to the reversible circuits described by Qiskit or QASM. Now if we simply change the direction of the data flow, the circuit will exactly run backwards.

In this way, more complicated propositional logic formula can be transformed to its reversible circuit version. Moreover, the conceptual reversible circuit has no explicit limitation on data types the wires can carry, and therefore it has the potential to express more advanced features without losing much readability.

2 Theseus

Since we can fill the gap between high-level languages and reversible circuits in some way, we can also design a high-level reversible langueges.

The relation between high-level reversible languages and reversible circuits is something like the relation between a traditional high-level language like Java or Racket and assembly or machine code. Therefore the circuit representation is not always easy to understand especially when the scale grows fairly large.

Theseus[1] is one of high-level programming languages for reversible computing. Its a typed language. Type isomorphisms preserve information that is required to run in the other direction, e.g.

```
expandBool :: Bool * a <-> a + a
| True, a <-> Left a
| False, a <-> Right a
```

It's a way to substitute the explicit boolean value (bit) with an implicit type label which preserves the imformation about which variant of the union type it belongs to.

For reversibly computing, there are cases that the value `a` continues to flow while the bit above `a` is no longer needed in current running direction, but still needed when running backwards. In these case, functions like `expandBool` works perfectly.

In reversible circuits, bits and wires carrying them cannot be temporarily hidden or generated halfway due to physical restrictions; matrices also cannot change their number of rows and columns between multiplications, because they must follow algebraic rules; but in terms of (high-level) programming languages, they are no longer restricted physically and algebraically: they can only be restricted logically. The extra dimension of freedom here is not only an advantage but also a challenge in terms of compilation and any other correctness preserving transformations.

3 Matrices

Matrices are natually point-free, but not natually reversible.

In **Part 1**, matrices are mentioned several times. This part will focus on which kind of matrices that can represent various components in reversible circuits, and their main properties.

Every matrix can represent a transformation (or map, function, operator...), but just like not every function is reversible, not every matrix can represent a component in reversible circuits.

In Qiskit, frequently used components like `cnot`, `ccx` and H gate have their matrix form. Besides, there are APIs like `operator()` that lets you program with matrix forms. These matrices are unitary:

$$\boldsymbol{A}^\dagger \boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^\dagger = \boldsymbol{I}$$

i.e.

$$\boldsymbol{A}^{-1} = \boldsymbol{A}^\dagger$$

Its relatively easier to understand why a matrix has to be invertible in order to represent a reversible gate: if not, the bits cannot flow in the other direction as shown in **Part 1** and will stuck.

For conjugate transpose, it corresponds to "swapping" the input and the output, which changes the direction of bit flow. Recall the properties of reversible gates, if the direction of bit flow changes, the function works exactly reversely, which matches the definition of inverse matrix.

Combining them will get the unitary matrix.

Every row and column of these kind of matrix represents a potential combinition of the input and output n bits. In this project, they have ascending order from left to right, and top to bottom.

$$\overset{\downarrow}{\leftarrow}\begin{pmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{pmatrix} \overset{\dagger}{\Longleftrightarrow} \overset{\uparrow}{\rightarrow}\begin{pmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{pmatrix}$$

$$\overset{\downarrow}{\leftarrow}\begin{pmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{pmatrix}(\boldsymbol{A}) \cdot \overset{\downarrow}{\leftarrow}\begin{pmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{pmatrix}(\boldsymbol{A}^\dagger) = \overset{\downarrow}{\leftarrow}\begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix}(\boldsymbol{I})$$

The same rule for the bit vector: an n-bit vector always has 2^n length.

If not stated explicitly, in the next parts of this project, the term "matrix" refers to unitary matrix. For some local gate shown below, in order to represent Bool^n types, there will be *active bits* to differentiate among various types, a circuit gate with a bits input and b bits ouput is always a $2^{\max(a,b)} \times 2^{\max(a,b)}$ matrix.

Globally viewing the entire circuit as a huge gate, the number of rows has to be equal to the number of columns. That is exactly the case when discussing about the set of $n \times n$ unitary matrices equipped with multiplication forms a monoid.

4 Compiling Theseus to Circuits

A subset of Theseus will be compiled to reversible circuits described by Qiskit or QASM in this part. Meanwhile, the matrix form will also be presented or discussed if it is intuitive and clear enough.

Here "compilation" is not as specific as the familiar process of correctness preserving transformation from a high-level language to the machine code, step by step. For example, parsing a piece of Theseus code into its abstract syntax tree form is not concerned in this project. The goal is trying to find the logical correspondence of any features in Theseus, question like "what kind of structure does conditional expression in Theseus correspond to in reversible circuits?" will be elaborated. In other words, this project interested in 2 endpoints of the whole compilation pass pipeline: Theseus code and circuits structure.

4.1 Data Representation

In Theseus, the type matters. One can define recursive types like natural number and binary tree. However, circuits only have the boolean type, which means after compilation all types of data in Theseus will have types $\text{Bool}^n, n \in \mathbb{N}^+$. It's not strange because for traditional high-level languages with or without a type system, when they are compiled to machine code, the case is always the same.

The "type system" of C language gives each "type" a hidden length of bits, and in this project, types will be handled like this. In other words, type denoted by a simbol, e.g. `a`, will hold *a active bits* in reversible circuits. This is a simplified denotation handy for demonstration, and the global matrix is always square. The pattern $2^{\max(x,y)} \times 2^{\max(x,y)}$ will appear frequently in the following discussion.

Take `Nat` as an example, one representation of type `Nat` is binary bounded integer:

```
type Nat5 = Bool * Bool * Bool * Bool * Bool
```

This defines a type of 5-bit natural number ranged among $[0, 2^5]$. Any functions defined over `Nat4` will finally refer to its definition. Such definition doesn't reflect the recursive nature as Peano's does, but it doesn't matter much in circuit level representation.

4.2 Function

In general, a "function" in reversible circuits can always be represented by a unitary matrix, as discussed in **Part 3**. What is interesting about function in Theseus is how can we compose those reversible function, and what is the corresponding operations (i.e. how to place the gates) in terms of circuits.

Parametrized maps are heavily used, since its a very natural way to reflect the idea of composing functions.

4.2.1 Converting to the reversible U_f

By applying the generalized form of reversible logical gates discussed in **Part 1**, i.e.

$$U_f(h, b_1, ..., b_n) = (f(b_1, ..., b_n) \oplus h, b_1, ..., b_n,)$$

we can convert a irreversible $n \to 1$ function f to its reversible version U_f.

Consider an example of AND gate: the irreversible function is

b_1	b_2	and(b_1, b_2)
0	0	0
0	1	0
1	0	0
1	1	1

The reversible AND gate is

$$U_{\text{and}}(h, b_1, b_2) = (\text{and}(b_1, b_2) \oplus h, b_1, b_2)$$

It's matrix form is

$$\left(\begin{array}{cccc|cccc} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & & & & & 1 \\ \hline & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \\ & & & 1 & & & & \end{array}\right)$$

The left (colored) half is what we should focus on when the first bit, $h = 0$:

- The yellow quarter means: the 2nd AND the 3rd bits yields 0.
- The blue quater means: the 2nd AND the 3rd bits yields 1.

This result exactly matches the irreversible function (three 0s, one 1). The right half is the same because the computation taken place at the first bit is and$(b_1, b_2) \oplus h$, the $\oplus$ operation here will "swap" the result of mapping, and thus making the matrix orthogonal.

4.2.2 Adjoint

Each reversible function has an adjoint:

```
adjoint :: f:(a <-> b) -> b <-> a
| (f a) <-> a
```

The semantics of composition combinators is:

$$\frac{c\, v_1 \mapsto v_2}{(\text{adjoint}\, c)\, v_2 \mapsto v_1}$$

The matrix form is to take its conjugate transpose:

$$\boldsymbol{A}\boldsymbol{v}_1 = \boldsymbol{v}_2 \quad \boldsymbol{A}^\dagger \boldsymbol{v}_2 = \boldsymbol{v}_1$$

Its circuit form is the "mirror image" with left-right reversed:

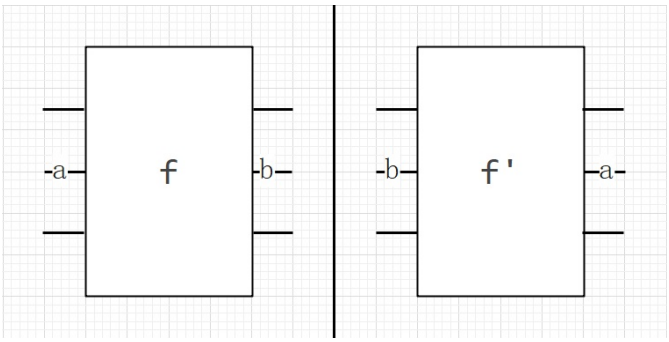

Figure 7. adjoint `f` (symmetric)

The two gates, representing f and its adjoint $f^\dagger$ respectively, are symmetric, and it's a vivid presentation of why the constructor `adjoint` also named `sym`.

In practice, we can just connect the wires inversely to the same gate, or flip the gate around to obtain its adjoint.

By definition, if a gate and its adjoint is connected adjacent to each other, it produces the identity gate:

$$\begin{aligned} \boldsymbol{A}\boldsymbol{A}^\dagger \boldsymbol{v}_1 &= \boldsymbol{v}_1 \\ \boldsymbol{A}\boldsymbol{A}^\dagger &= \boldsymbol{I} \end{aligned}$$

That is intuitive as in quantum programming, a symmetric gate (for gate like `not`, `cnot` and `ccx`, a symmetric gate means a same gate) is always placed to neutralize some side effects.

4.2.3 Function Composition

In Theseus, function composition is a parametrized map:

```
_._ :: f:(a <-> b) -> g:(b <-> c) -> a <-> c
| a <-> g (f a)
```

The semantics of composition combinators is:

$$\frac{c_1\, v_1 \mapsto v \quad c_2\, v \mapsto v_2}{(c_1 \circ c_2)\, v_1 \mapsto v_2}$$

Its matrix form is the multiplication of two matrices:

$$\begin{aligned} \boldsymbol{A} \cdot \boldsymbol{v}_1 &= \boldsymbol{v} \\ \boldsymbol{B} \cdot \boldsymbol{v} &= \boldsymbol{v}_2 \\ (\boldsymbol{B}\boldsymbol{A}) \cdot \boldsymbol{v}_1 &= \boldsymbol{v}_2 \end{aligned}$$

where $\boldsymbol{A}$ is a $2^{\max(a,b,c)} \times 2^{\max(a,b,c)}$ matrix representing `f`, and $\boldsymbol{B}$ is a $2^{\max(a,b,c)} \times 2^{\max(a,b,c)}$ matrix representing `g`.

The circuit from is like:

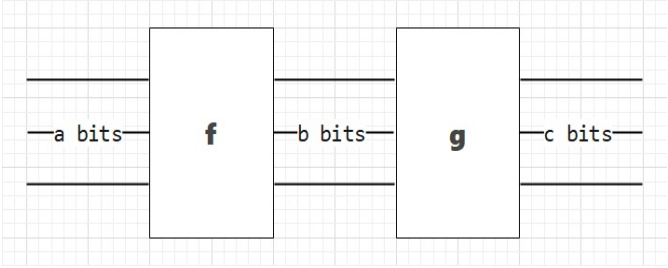

Figure 8. circuit for function composition, with a *active bits* of input and c *active bits* of output

Composition corresponds to connecting two gates physically. This circuit needs at least $\max(a, b, c)$ bits initialized to carry the input/output values.

4.2.4 Parametrized Map: Product

```
_*_ :: f:(a <-> b) -> g:(c <-> d) -> a * c <-> b * d
| (a, c) <-> (f a, g c)
```

The semantics of composition combinators is:

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

The matrix form is not so clear to show. It's a $2^{\max(a+c,b+d)} \times 2^{\max(a+c,b+d)}$ matrix, probably very large. But its circuit form is rather easy to fathom:

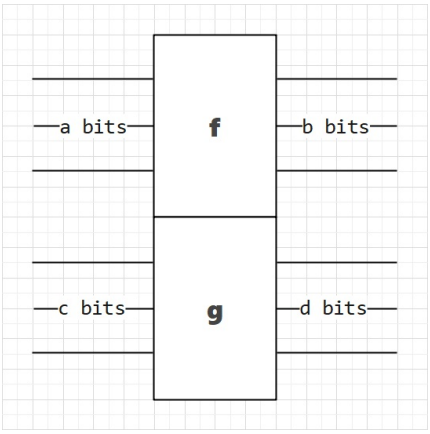

Figure 9. circuit for product with parametrized map, with $a + c$ *active bits* of input and $b + d$ *active bits* of output

Product with parametrized map corresponds to placing two gates "vertically". This circuit needs at least $\max(a + c, b + d)$ bits initialized to carry the input/output values.

4.2.5 Parameterized Map: Sum*

```
_+_ :: f:(a <-> b) -> g:(c <-> d) -> a + c <-> b + d
| Left a <-> Left (f a)
| Right c <-> Right (g c)
```

The semantics if composition combinators is:

$$\frac{c_1\, v_1 \mapsto v_2 \quad c_2\, v_3 \mapsto v_4}{(c_1 \oplus c_2)(\text{left}\, v_1) \mapsto \text{left}\, v_2 \quad (c_1 \oplus c_2)(\text{right}\, v_3) \mapsto \text{right}\, v_4}$$

The sum has 2 patterns, but the data can only match one of them. The parallelism here is a bit tricky, because pattern matching matters now, and the circuit has no tags or labels to differentiate them. In general, pattern matching can be viewed as 2 atomic operations: first test, then bind. In Theseus, the matched pattern and the expression is the same thing.

It's *either* a $2^{\max(a,b)} \times 2^{\max(a,b)}$ matrix (`f`) *or* a $2^{\max(c,d)} \times 2^{\max(c,d)}$ matrix (`g`).

It's circuit form is temporarily conceptual:

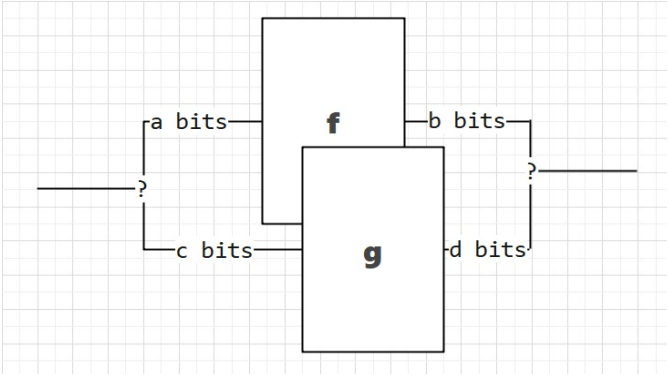

Figure 10. conceptual circuit for sum with parametrized map

The two `?` here means there should be some ways to differentiate between 2 tags of the union types.

4.3 Conditional Expression

In traditional programming languages, an if-expression acts as a demux: asking a question and branching the program. In Theseus, the idea is similar while the answer to the question must be remembered in order to maintain reversibility, and the conditional expression is actually a parametrized map.

4.3.1 General If-Expression

Take `cnot` as an example. `cnot` is one of the most simple gate that can be expressed by conditional expression in Theseus:

```
cnot :: Bool * Bool <=> Bool * Bool
| ctrl, bit <=> if ~th:not ~el:id
```

Or simply `cnot = if ~th:not ~el:id`.

Its matrix form is

$$\left(\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right)$$

Devide it into 4 parts and focus on the 2 colored quarters:

- The yellow quarter means: if the control bit is 0, then apply $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ to the 2nd bit.
- The blue quarter means: if the control bit is 1, then apply $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ to the 2nd bit.

Notice that $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ are exactly the `id` and `not` respectively.

Follow this obeservation, the Toffoli gate `ccx`'s matrix form is

$$\begin{pmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & & 1 \\ & & & & & & 1 & \end{pmatrix}$$

The "topological structure" of this matrix is quite similar to the Toffoli gate implemented in the graphical language[2, Sec 2.1], though matrix doesn't have anything like types.

It can be expressed by the following Theseus code:

```
ccx = if ~th:[if ~th:not ~el:id(1)] ~el:id(2)
```

It's a nested parametrized map. The code inside `[]` is `cnot`, so the full `ccx` definition is:

```
ccx :: Bool * Bool * Bool <-> Bool * Bool * Bool
| c1, c2, bit <-> if ~th:cnot ~el:id
```

In this way, we can recursively build higher order Toffoli gates.

Generally, for conditional expression in Theseus:

```
if :: th:(a <-> b) -> el:(a <-> b) -> Bool * a <-> Bool * b
| True, x <-> True, (th x)
| False, x <-> False, (el x)
```

Its matrix form can be expressed as

$$\begin{pmatrix} \boldsymbol{E} & \\ & \boldsymbol{T} \end{pmatrix}$$

where $\boldsymbol{E}$ represents the function inlined in `else` branch, and $\boldsymbol{T}$ represents the function inlined in `then` branch. Both $\boldsymbol{E}$ and $\boldsymbol{T}$ are $2^{\max(a,b)} \times 2^{\max(a,b)}$ matrix.

Its circuit form is like

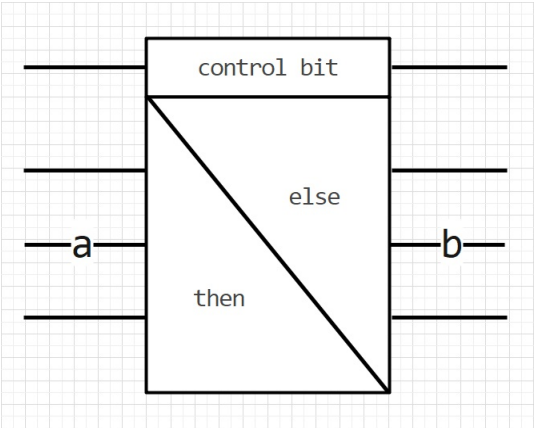

Figure 11. circuit for conditional expression

Two functions `then` and `else` are hardcoded while classically only one of them will be activated, according to what value passes through the control wire.

4.3.2 Nested conditional expression

As discussed above, `ccx` and higher-order Toffoli gates are all nested conditional expressions.

In general, we can nest the pattern $\left(\begin{array}{c|}\boldsymbol{E}\\\hline\boldsymbol{T}\end{array}\right)$ in both branches to arbitrary heights:

$$\left(\begin{array}{cc}\boldsymbol{E}_0 & \\ & \left(\begin{array}{cc}\boldsymbol{E}_1 & \\ & \boldsymbol{T}_1\end{array}\right)\end{array}\right)$$

$$\left(\begin{array}{cc}\left(\begin{array}{cc}\boldsymbol{E}_0 & \\ & \boldsymbol{T}_0\end{array}\right) & \\ & \left(\begin{array}{cc}\boldsymbol{E}_1 & \\ & \boldsymbol{T}_1\end{array}\right)\end{array}\right)$$

$$\left(\begin{array}{cc}\left(\begin{array}{cc}\left(\begin{array}{cc}\cdots & \\ & \boldsymbol{T}_{00}\end{array}\right) & \\ & \boldsymbol{T}_0\end{array}\right) & \\ & \left(\begin{array}{cc}\boldsymbol{E}_1 & \\ & \left(\begin{array}{cc}\boldsymbol{E}_{11} & \\ & \cdots\end{array}\right)\end{array}\right)\end{array}\right)$$

5 Summary

Part 1 shows with a tiny example the motivation to fill the gap between high-level languages and reversible circuits, by demonstrating the idea

Part 2 briefly introduces a reversible high-level language, Theseus. **Part 3** mainly discusses what and how matrices can represent reversible gates, and lays foundation for next parts.

Part 4 mainly explores how can a subset of Theseus be compiled into reversible circuits. While most features like functions, parametrized maps are elaborated, there are still some important features not covered in this part, like iterations and *trace* operator.

References

1. Roshan P. James and Amr Sabry. Theseus: A High Level Language for Reversible Computing. 2014.

2. Roshan P. James and Amr Sabry. Isomorphic interpreters from logically reversible abstract-machines. In Reversible Computation, 2012.