Fractional Types

Expressive and Safe Space Management for Ancilla Bits

Anonymous Author(s)

Abstract

In reversible and quantum computing, the management of space is subject to two broad classes of constraints. First, as is the case for general-purpose computation, every allocation must be paired with a matching de-allocation. Second, space can only be safely de-allocated if its contents are restored to their initial value from allocation time. Generally speaking, the state of the art provides limited partial solutions that address the first constraint by imposing a stack discipline and by leaving the second constraint to programmers' assertions.

We propose a novel approach based on the idea of *fractional types.* As a simple intuitive example, allocation of a new boolean value initialized to false also creates a value 1/false that can be thought of as a garbage collection (gc) process specialized to reclaim, and only reclaim, storage containing the value false. This gc process is a first-class entity that can be sliced and diced by combining it with other gc processes and decomposing it in smaller processes.

Technically, we formalize this idea in the context of a reversible language founded on type isomorphisms, prove its fundamental correctness properties, and illustrate its expressiveness using a wide variety of examples. The development is backed by a fully-formalized Agda implementation.

1 Introduction

If quantum field theory is correct (as it so far seems to be) then information, during any physical process, is neither created nor destroyed. Landauer [13, 35, 36], Bennett [11, 12, 14], Fredkin [25] and others made compelling arguments that this physical principle induces a corresponding computational principle of "conservation of information." This principle is indeed one of the defining characteristics of quantum computing and its classical restriction known as reversible computing.

Quantum and Reversible Computing Based on Type Isomorphisms. The Toffoli gate is known to be universal for classical reversible circuits [55]. Adding just one gate (the Hadamard gate) produces a universal set of primitives for quantum circuits [7]. The "only" difference between the two circuit models is that quantum circuits can process superpositions of values (waves) in one step whereas classical circuits lack this form of parallelism. Most importantly, structurally the two circuit models are identical and one can derive properties valid for both families by focusing on the simpler classical model. In fact, classical reversible computations are often used as "subroutines" of quantum computations, i.e., classical reversible circuits can be applied to either classical inputs or to quantum inputs.

Instead of using the low-level circuit model of computation, one can leverage the full force of type theory and category theory by expressing reversible classical computations as *isomorphisms over finite types* [22, 30] or *equivalences over groupoids* [17]. This perspective is similarly universal for reversible computation (and can be extended to quantum computation [47]) but with the advantage of exposing a rich mathematical structure in reversible computations that we will exploit to solve the "ancilla problem" explained next.

Temporary Storage using Ancilla Bits. The universality of the Toffoli gate for classical reversible computing and the universality of the Toffoli-Hadamard gates for quantum computing come with a catch and should not distract from efficiency and safety concerns. The theorems proving universality (i) assume that temporary storage (often called *ancilla bits*) may be used [55], and (ii) that this temporary storage is returned to its initial state before de-allocation. Indeed if no temporary storage is allowed, the Toffoli gate is not universal [1, 59]. And, as would be expected, the more temporary storage is allowed, the more efficient certain computations can become.

More fundamentally, the condition requiring that the temporary storage is only de-allocated when returned to its initial condition is a safety condition. Violating this condition destroys the symmetry between input and output, making the circuits irreversible and, in the quantum model, causes decoherence that may destroy the quantum state. As reviewed in Sec. 2, ancilla bits have a number of other critical applications, yet are poorly supported in current reversible and quantum programming languages making them a common source of bugs.

Conservation of Information and Negative Entropy. The conventional theory of communication [51] views a type with N values as an abstract system that has N distinguishable states; the amount of information contained in each state is $\log N$. This entropy is a measure of information which materializes itself in memory or bandwidth requirements

PL'18, January 01–03, 2018, New York, NY, USA
2018.

when storing or transmitting elements of this type. Thus a type with 8 elements needs 3 bits of memory for storage or 3 bits of bandwidth for communication. The information contained in a composite state, by definition of the logarithm, is the sum of the information contained in its constituents. For example, the type $A \times B$ where A has two elements and B has three elements can be thought of a composite system consisting of two independent unrelated subsystems. Each state of the composite system therefore contains $\log(2 * 3) = \log 2 + \log 3$ bits, the sum of the information contained in each subsystem. If we could imagine a *fractional type* $\frac{1}{A}$, this type would introduce negative entropy. For example, a type with "cardinality" $\frac{1}{8}$ has entropy $\log \frac{1}{8} = -3$. It is natural to interpret this negative entropy just like we interpret "negative money," as a resource (space or bandwidth) to be repaid (reclaimed) by some other part of the system. Indeed, we will introduce such fractional types and use them to represent "garbage collection processes" that reclaim temporary storage. Just like in the case of "negative money" it is critical that these fractional types be linearly used, i.e., they cannot be duplicated or erased: this property is automatically enforced in a reversible computational model.

Outline. We solve the ancilla problem in reversible and quantum computation using a novel concept: *fractional types.* In the next section, we introduce the problem of ancilla, motivate its importance, and explain the limitations of current solutions. In Sec. 3, we review existing work [16, 17, 30, 31] that introduces a reversible programming language built using type isomorphisms. Although the concept of fractional types might apply to general-purpose languages (after some adaptation), its natural technical definition exploits symmetries present in the categorical model of type isomorphisms. We present such a definition in Sec. 4 which captures and illustrates the main novelties of fractional types and their runtime representations as first-class gc processes. This definition allows allocation and de-allocation of ancillae in patterns beyond the scope model but, like existing stack-based solutions, still requires a runtime check to verify the safety of de-allocation. To make the arguments regarding memory management explicit, Sec. 5 presents an abstract machine with a heap component that grows and shrinks as ancillae are allocated and de-allocated via garbage collection. In Sec. 6 we lift "plain" programs manipulating ancillae to the type system in order to exploit the power of dependent types to guarantee, for the first time, the condition necessary for safe de-allocation of ancillae. Technically this solution requires the introduction of pointed types on top of which we define both a monad and comonad of singleton types. The tracking of ancillae value is done within the type system via singleton types. A number of examples exploiting the singleton-based type system are presented to demonstrate the full expressiveness of reasoning about fractional types. The type safety proofs demonstrate that plain (but guaranteed safe) programs can be extracted and executed without errors or runtime checks in the conventional type system. The last section concludes with a discussion of related work and a summary of our results.

The paper includes a number of languages that express different points. The three most important languages and their relationships are highlighted in this diagram:

$$\begin{tikzcd}
\Pi/D \text{ (Sec. 4.4)} & & \Pi/\bullet \text{ (Sec. 6.2)} \arrow[ll, swap, "\substack{\text{(Sec. 6.3)} \\ \text{Extracting safe programs}}"] \\
& \Pi \text{ (Sec. 3)} \arrow[ul, "\text{+ fractionals / dynamic check}"] \arrow[ur, swap, "\text{+ pointed / singleton / safe fractionals}"] &
\end{tikzcd}$$

The language Π is the base language of type isomorphisms. It is defined in Sec. 3, and first extended with fractional types in Sec. 4.4. This extension Π/D requires a runtime check to ensure safe de-allocation. In Sec. 6.2, Π is independently extended to $\Pi/\bullet$ which uses singleton types to keep track of the value of ancillae. In Sec. 6.3, we show that Π/D programs extracted from $\Pi/\bullet$ are safe without the runtime checks associated with de-allocation.

2 Ancilla Bits

Restricting a reversible (classical or quantum) circuit to use no ancillae is like restricting a Turing machine to use no memory other than the n bits used to represent the input [1]. As such a restriction disallows countless computations for trivial reasons, reversible and quantum models of computation have, since their inception, included management for scratch storage in the form of ancilla bits [55] with the fundamental restriction that such bits must be returned to their initial states before being safely reused or de-allocated.

2.1 Applications

Beyond computability and efficiency reasons, ancillae have found a wide range of interesting applications.

Quantum blackboxes and Phase Kickback Trick. Consider a small database with four elements a, b, c, and d. We are given a function f that maps every element to 0 except for one element of interest, e.g., $f(a) = 1$ and $f(b) = f(c) = f(d) = 0$. In the worst case, finding the element of interest might require applying f to every element of the database. In the quantum world we can find the element much more efficiently using Grover's algorithm [27]. We start by building an equally weighted superposition of the elements: $\frac{1}{2}|a\rangle + \frac{1}{2}|b\rangle + \frac{1}{2}|c\rangle + \frac{1}{2}|d\rangle$ and operate concurrently of the superposition as follows:

$$\tfrac{1}{2}|a\rangle + \tfrac{1}{2}|b\rangle + \tfrac{1}{2}|c\rangle + \tfrac{1}{2}|d\rangle$$
$$\Rightarrow \text{phase kickback}$$
$$-\tfrac{1}{2}|a\rangle + \tfrac{1}{2}|b\rangle + \tfrac{1}{2}|c\rangle + \tfrac{1}{2}|d\rangle$$
$$\Rightarrow \text{diffusion}$$
$$(\tfrac{1}{2} - (-\tfrac{1}{2}))|a\rangle + (\tfrac{1}{2} - \tfrac{1}{2})|b\rangle + (\tfrac{1}{2} - \tfrac{1}{2})|c\rangle + (\tfrac{1}{2} - \tfrac{1}{2})|d\rangle)$$
$$\Rightarrow \text{simplification}$$
$$|a\rangle$$

The question, which arises repeatedly in quantum computation, is how to implement operations such as phase kickback. Essentially we want to concurrently perform the following action on every element of the superposition:

if *current-element* = $|a\rangle$
then *negate*
else *identity*

In the quantum world, naïvely attempting to test whether the current element is equal to $|a\rangle$ by performing a measurement would destroy the quantum superposition, defeating the entire algorithm.

Ancilla bits offer a solution. In the words of Matuschak and Nielsen [40], the idea is as follows:

> There's a rough heuristic worth noting here, which is that you can often convert `if-then` style of thinking into quantum circuits. You introduce an ancilla qubit to store the outcome of evaluating the `if` condition. And then depending on the state of the ancilla, you perform the appropriate state manipulation. Finally, when possible you reverse the initial computation, resetting the ancilla to its original state so you can subsequently ignore it.

Quantum Error Correction. Quantum error correction requires the ability to replicate the state of a qubit onto a number of qubits [41, Ch. 3]. This is impossible to do directly, by the no-cloning theorem [20, 45, 58]. Again ancilla qubits solve the problem: because they start in a known initial state, it is possible to create a simple circuit that makes their output state match a protected qubit without knowing or disturbing the protected qubit. Chong et al. [24] illustrate the idea with a simple example:

> Consider a 3-qubit quantum majority code in which a logical "0" is encoded as "000" and a logical "1" is encoded as "111." Just as with a classical majority code, a single bit-flip error can be corrected by restoring to the majority value. Unlike a classical code, however, we cannot directly measure the qubits else their quantum state will be destroyed. Instead, we measure syndromes from each possible pair of qubits by interacting them with an ancilla, then measure each ancilla. Although the errors to the qubits are actually continuous, the effect of measuring the ancilla is to discretize the errors, as well as inform us whether an error occurred so that it can be corrected.

Deferring Quantum Measurements. The evolution of a quantum system is deterministic. Measurement adds significant complications, collapsing the quantum state, and producing probabilistic results. However, using ancilla qubits, it is possible to avoid doing any intermediate measurements in a quantum circuit [19].

2.2 Ancillae Bugs in Benchmarks.

Despite its fundamental importance, the management of ancillae is poorly supported in current languages, simulators, and toolkits. A recent analysis of bugs in quantum benchmarks [28] demonstrates this. The analysis reveals that lack of language support in the management of ancillae leads to several problems. For example:

> **Bug type 6: Incorrect composition of operations using mirroring** Section 4.5 discussed how bugs in de-allocating ancillary qubits can happen due to bad parameters. Here we see how bugs in de-allocating ancillary qubits can happen due to incorrect composition of operations following a mirroring pattern. For example, in Table 7, the operations in rows 2 and 3 are respectively mirrored and undone in rows 6 and 5. These lines of code need careful reversal of every loop and every operation.

3 Preliminaries: Π

The step from reversible classical circuits to quantum circuits requires the addition of just one gate that creates superpositions. Similarly the step from a reversible classical programming language to a quantum one requires just the management of superpositions. For example, starting from the classical reversible language Theseus [32], a small extension produces a quantum language with quantum control [47]. Theseus [32] is a language that layers convenient syntactic extensions over an even more basic language Π of combinators witnessing isomorphisms over finite types [17]. The combinator-based language has the advantage of being more amenable to formal analysis for at least two reasons: (i) it is conceptually simple and small, and (ii) it has direct and evident connections to type theory and category theory. Indeed our solution for managing ancillae is inspired by the construction of *compact closed categories* [33]. These categories extend the monoidal categories [9, 10, 39] which are used to model many resource-aware (e.g., based on linear types) programming languages [15, 34] (including Π) with

$$
\begin{array}{rrcll}
\mathrm{id}\!\leftrightarrow: & \tau & \leftrightarrow & \tau & : \mathrm{id}\!\leftrightarrow \\[1em]
\mathrm{unite}_{+}\mathrm{l}: & 0+\tau & \leftrightarrow & \tau & : \mathrm{uniti}_{+}\mathrm{l} \\
\mathrm{swap}_{+}: & \tau_1+\tau_2 & \leftrightarrow & \tau_2+\tau_1 & : \mathrm{swap}_{+} \\
\mathrm{assocl}_{+}: & \tau_1+(\tau_2+\tau_3) & \leftrightarrow & (\tau_1+\tau_2)+\tau_3 & : \mathrm{assocr}_{+} \\[1em]
\mathrm{unite}_{*}\mathrm{l}: & 1\times\tau & \leftrightarrow & \tau & : \mathrm{uniti}_{*}\mathrm{l} \\
\mathrm{swap}_{*}: & \tau_1\times\tau_2 & \leftrightarrow & \tau_2\times\tau_1 & : \mathrm{swap}_{*} \\
\mathrm{assocl}_{*}: & \tau_1\times(\tau_2\times\tau_3) & \leftrightarrow & (\tau_1\times\tau_2)\times\tau_3 & : \mathrm{assocr}_{*} \\[1em]
\mathrm{absorbr}: & 0\times\tau & \leftrightarrow & 0 & : \mathrm{factorzl} \\
\mathrm{dist}: & (\tau_1+\tau_2)\times\tau_3 & \leftrightarrow & (\tau_1\times\tau_3)+(\tau_2\times\tau_3) & : \mathrm{factor}
\end{array}
$$

$$\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \qquad \vdash c_2 : \tau_2 \leftrightarrow \tau_3}{\vdash c_1 \mathbin{⨾} c_2 : \tau_1 \leftrightarrow \tau_3}$$

$$\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \qquad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \oplus c_2 : \tau_1 + \tau_3 \leftrightarrow \tau_2 + \tau_4}$$

$$\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \qquad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \otimes c_2 : \tau_1 \times \tau_3 \leftrightarrow \tau_2 \times \tau_4}$$

Figure 1. Π-terms and combinators.

a new type constructor that creates duals or inverses to existing types. This dual will be our fractional type.

3.1 Core Π

The syntax of the language consists of several categories:

Value types	t	$::=$	$0 \mid 1 \mid t+t \mid t\times t$
Values	v	$::=$	$\mathrm{tt} \mid \mathrm{inj}_1(v) \mid \mathrm{inj}_2(v) \mid (v,v)$
Program types			$t \leftrightarrow t$
Programs	c	$::=$	(See Fig. 1)

Focusing on finite types, the building blocks of type theory are: the empty type (0), the unit type (1), the sum type (+), and the product (×) type. One may view each type A as a collection of physical wires that can transmit $|A|$ distinct values where $|A|$ is the size of a type, computed as: $|0| = 0$; $|1| = 1$; $|A + B| = |A| + |B|$; and $|A \times B| = |A| * |B|$. Thus the type $\mathbb{B} = 1 + 1$ corresponds to a wire that can transmit two values, i.e., bits, with the convention that $\mathrm{inj}_1(\mathrm{tt})$ represents false and $\mathrm{inj}_2(\mathrm{tt})$ represents true. The type $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$ corresponds to a collection of wires that can transmit three bits. From that perspective, a type isomorphism between types A and B (such that $|A| = |B| = n$) models a *reversible* combinational circuit that *permutes* the n different values. These type isomorphisms are collected in Fig. 1. It is known that these type isomorphisms are sound and complete for all permutations on finite types [22, 23] and hence that they are *complete* for expressing combinational circuits [25, 30, 55]. Algebraically, these types and combinators form a *commutative semiring* (up to type isomorphism). Logically they form a superstructural logic capturing space-time tradeoffs [52]. Categorically, they form a *distributive bimonoidal category* [37].

3.2 Small Examples

Below, we show code that defines types corresponding to bits (booleans), two-bits, three-bits, and four-bits. We then define an operator that builds a controlled version of a given combinator c. This controlled version takes an additional "control" bit and only applies c if the control bit is true. The code then iterates the control operation several times starting from boolean negation.

```
𝔹 𝔹² 𝔹³ 𝔹⁴ : 𝕌
𝔹  = 𝟙 +ᵤ 𝟙
𝔹² = 𝔹 ×ᵤ 𝔹
𝔹³ = 𝔹 ×ᵤ 𝔹²
𝔹⁴ = 𝔹 ×ᵤ 𝔹³

ctrl : {A : 𝕌} → (A ↔ A) → 𝔹 ×ᵤ A ↔ 𝔹 ×ᵤ A
ctrl c = dist ⨾ (id↔ ⊕ (id↔ ⊗ c)) ⨾ factor

NOT : 𝔹 ↔ 𝔹
NOT = swap₊

CNOT : 𝔹² ↔ 𝔹²
CNOT = ctrl NOT

TOFFOLI : 𝔹³ ↔ 𝔹³
TOFFOLI = ctrl (ctrl NOT)

CTOFFOLI : 𝔹⁴ ↔ 𝔹⁴
CTOFFOLI = ctrl (ctrl (ctrl NOT))
```

By construction, the TOFFOLI function takes three bits, and negates the third if the first two are both true. As Toffoli [55] demonstrated, this function is universal for classical reversible circuits assuming it is possible to allocate and de-allocate ancilla bits. Indeed without using ancilla bits, it is not possible to use compositions of TOFFOLI functions to implement more general versions that are controlled by more than two bits (for example, CTOFFOLI). The proof is simple: the TOFFOLI function depends on two control bits and possibly negates a third; when used in a circuit with four bits, the unused bit can be either state, which means that TOFFOLI generates an even permutation. The function CTOFFOLI is however an odd permutation which is unreachable from even permutations. This function will however be definable using composition of TOFFOLI once we introduce ancilla bits in the next section.

4 Interchangeable Ancillae

The management of ancilla data can be teased into two relatively distinct subproblems:

- Every allocated ancilla bit must be de-allocated, and
- De-allocation of an ancilla bit is only safe if the bit is restored to its allocation-time initial value.

In this section, we focus on the first subproblem and address the second in later sections.

4.1 Scoped Ancilla

A conventional and efficient way to ensure that every allocation is paired with a matching de-allocation is to impose a stack discipline. An illustrative demonstration of this idea is the `with_ancilla` operator from Quipper [26] :

```
with_ancilla :: (Qubit -> Circ a) -> Circ a
```

The operator takes a block of gates parameterized by an ancilla, allocates a new ancilla of type `Qubit` initialized to $|0\rangle$, and runs the given block of gates. At the end of its execution, the block is expected to return the ancilla to the state $|0\rangle$ (this is the second subproblem addressed later) at which point it is de-allocated. In this scoped-by-construction approach, allocation and de-allocation of ancillae requires nothing beyond conventional parameter-passing techniques in which the parameter is allocated before entry to a function and de-allocated at exit.

This scoped model is therefore quite a pragmatic choice. It is however limited, and to understand its limitations more vividly, we propose the following analogy: allocating an ancilla by creating a new wire in the circuit is like borrowing some money from a global external entity (the memory manager); the computation has access to a new resource temporarily. De-allocating the ancilla is like returning the borrowed money to the global entity; the computation no longer has access to the additional resource. It would however be unreasonably restrictive to insist that the person (function) borrowing the money must be the same person (function) returning it. Indeed, as far as reversible or quantum computation is concerned, the only important invariant is that information is conserved, i.e., that money is conserved. The identities of bits or qubits are not observable as they are all interchangeable in the same way that particular bills with different serial numbers are interchangeable in financial transactions. Thus the only invariant is that the net flow of money between the computation and the global entity is zero. This observation allows us to go even further than just switching the identities of borrowers. It is even possible for one person to borrow $10, and have three different persons collectively collaborate to pay back the debt with one person paying $5, another $2, and a third $3.

Computationally, this extra generality is not a gratuitous concern: since scope is a *static property* of programs, it does not allow the flexibility of heap allocation in which the lifetime of resources is dynamically determined. To summarize, since the reversible-quantum computational framework guarantees that information is preserved, it also permits fascinating mechanisms in which allocations and de-allocations can be sliced and diced, decomposed and recomposed, run forwards and backwards, in arbitrary ways as long as the net balance is 0.

4.2 GC Value of Fractional Type

Instead of restricting ancillae to be scoped, we separate allocation and de-allocation into two dual actions η and ϵ whose types are (almost):

$$\eta_A : 1 \leftrightarrow A \times 1/A : \epsilon_A$$

The names and types of these operations are inspired by compact closed categories which are extensions of the monoidal categories that model Π. Intuitively, η allows one from "no information" to create a pair of a value of type A and a value of type $1/A$. We interpret the latter value as a gc process specialized to collect the created value. Dually, ϵ applies the gc process to the appropriate value annihilating both.[1]

To make this idea work, several technical details are needed. Most notably, we must exclude the empty types from this creation and annihilation process. Otherwise, we would be able to prove that:

$$\begin{array}{rcll} 1 & = & 0 \times 1/0 & \text{by } \eta_0 \\ & = & 0 & \text{by absorbr} \end{array}$$

The second important detail is to ensure that the gc process is specialized to collect a particular value. We however defer this point to the next section and for now create gc processes that might be applied to incorrect values throwing exceptions instead of collecting them. We therefore extend the syntax of core Π in Sec. 3.1 as follows:

$$\begin{array}{llcl} \textit{Value types} & t & ::= & \cdots \mid 1/t \\ \textit{Non-empty types} & t_\bullet & & \\ \textit{Values} & v & ::= & \cdots \mid \circlearrowright \\ \textit{Program types} & & & t \leftrightarrow t \\ \textit{Programs} & c & ::= & \cdots \mid \eta_{t_\bullet} : 1 \leftrightarrow (t_\bullet \times 1/t_\bullet) \\ & & & \mid \epsilon_{t_\bullet} : (t_\bullet \times 1/t_\bullet) \leftrightarrow 1 \end{array}$$

The metavariable $t_\bullet$ excludes all types that are equivalent to the empty type, e.g., $(1+1) \times 0$. For now, all gc processes are represented at runtime using a trivial value denoted by $\circlearrowright$.

In the Agda formalization, the types of η and ϵ are:

```
η : {t : 𝕌} {t≠0 : ¬ card t ≡ 0} ⟶ 𝟙 ↔ t ×ᵤ 𝟙/ t
ε : {t : 𝕌} {t≠0 : ¬ card t ≡ 0} ⟶ t ×ᵤ 𝟙/ t ↔ 𝟙
```

where the operations take an implicit argument asserting the given type is non-empty. A *big step operational semantics* for the language interprets each combinator (except η

[1] Another interesting interpretation is that these operations correspond to creation and annihilation of entangled particle/antiparticle pairs in quantum physics [44].

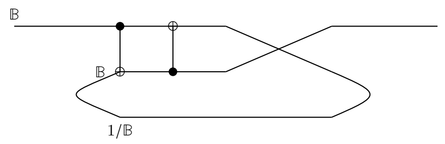
and ϵ) of type $t_1 \leftrightarrow t_2$ as a (reversible) function mapping values of type t_1 to values of type t_2 (and vice-versa).[2] Extending this interpreter to handle η and ϵ requires that the interpreter returns either a value of the appropriate type or an exception:

```
interp : {t₁ t₂ : 𝕌} ⟶ (t₁ ↔ t₂) ⟶ ⟦ t₁ ⟧ ⟶ Maybe ⟦ t₂ ⟧
interp swap⁎ (v₁ , v₂) = just (v₂ , v₁)
interp (c₁ ⨾ c₂) v = interp c₁ v »= interp c₂
-- (elided)
interp (η {t} {t≠0}) tt = just (default t {t≠0} , ○)
interp (ε {t} {t≠0}) (v' , ○) with v' ≟ᵤ (default t {t≠0})
... | yes _ = just tt
... | no _ = nothing
```

When ancillae are created, they are created with a specific default value depending on the type. When the gc process encounters a value to be collected, it checks whether this value is the default value for the type and if so collects it. Otherwise it returns nothing indicating an exception.

4.3 Examples

The first example below illustrates the basic functionality of ancillae. The Agda code is written in a style that reveals the intermediate steps for easier correspondence with the figure. The circuit has one input and one output. Immediately after receiving the input, the circuit generates an ancilla wire and its corresponding gc process (first two steps in the Agda definition). The original input and the ancilla wire interact using two CNOT gates, after which the ancilla wire is redirected to the output (next three steps in the Agda code). Finally the original input is gc'ed (last two steps in the Agda code). The entire circuit is extensionally equivalent to the identity function but it does highlight an important functionality beyond scoped ancillae management: the allocated ancilla bit is redirected to the output and a completely different bit (with the proper default value) is collected instead.



```
id' : 𝔹 ↔ 𝔹
id' =
  let η' = η {𝔹} {𝔹≠0}
      ε' = ε {𝔹} {𝔹≠0}
  in  𝔹
  ↔⟨ uniti⁎r ⟩                               𝔹 ×ᵤ 𝟙
  ↔⟨ id↔ ⊗ η'⟩                               𝔹 ×ᵤ (𝔹 ×ᵤ 𝟙/ 𝔹)
  ↔⟨ assocl⁎ ⟩                               (𝔹 ×ᵤ 𝔹) ×ᵤ 𝟙/ 𝔹
  ↔⟨ (CNOT ⨾ CNOT' ⨾ swap⁎) ⊗ id↔ ⟩ (𝔹 ×ᵤ 𝔹) ×ᵤ 𝟙/ 𝔹
  ↔⟨ assocr⁎ ⟩                               𝔹 ×ᵤ (𝔹 ×ᵤ 𝟙/ 𝔹)
  ↔⟨ id↔ ⊗ ε'⟩                               𝔹 ×ᵤ 𝟙
  ↔⟨ unite⁎r ⟩                               𝔹 □
```

[2]In Sec. 5 we will define a small step abstract machine that exposes the space used at each step.

```
idcheck : (b : ⟦ 𝔹 ⟧) ⟶ interp id' b ≡ just b
idcheck 𝔽 = refl
idcheck 𝕋 = refl
```

The second example illustrates the manipulation of gc processes. A process for collecting a pair of values can be decomposed into two processes each collecting one of the values (and vice-versa):

```
rev× : {A B : 𝕌} {A≠0 : ¬ card A ≡ 0} {B≠0 : ¬ card B ≡ 0} ⟶
       𝟙/ (A ×ᵤ B) ↔ 𝟙/ A ×ᵤ 𝟙/ B
rev× {A} {B} {A≠0} {B≠0} =
  let η₁ = η {A} {A≠0}
      η₂ = η {B} {B≠0}
      ε' = ε {A ×ᵤ B} {A×B≠0}
  in                  𝟙/ (A ×ᵤ B)
  ↔⟨ uniti⁎l ⨾ uniti⁎l ⨾ assocl⁎ ⟩
                      (𝟙 ×ᵤ 𝟙) ×ᵤ 𝟙/ (A ×ᵤ B)
  ↔⟨ (η₁ ⊗ η₂) ⊗ id↔ ⟩
                      ((A ×ᵤ 𝟙/ A) ×ᵤ (B ×ᵤ 𝟙/ B)) ×ᵤ 𝟙/ (A ×ᵤ B)
  ↔⟨ (shuffle ⊗ id↔) ⨾ assocr⁎ ⟩
                      (𝟙/ A ×ᵤ 𝟙/ B) ×ᵤ ((A ×ᵤ B) ×ᵤ 𝟙/ (A ×ᵤ B))
  ↔⟨ id↔ ⊗ ε'⟩
                      (𝟙/ A ×ᵤ 𝟙/ B) ×ᵤ 𝟙
  ↔⟨ unite⁎r ⟩
                      𝟙/ A ×ᵤ 𝟙/ B □
  where
    shuffle : {A B C D : 𝕌} ⟶ (A ×ᵤ B) ×ᵤ (C ×ᵤ D) ↔ (B ×ᵤ D) ×ᵤ (A ×ᵤ C)
```

4.4 Choosing Ancillae Values: Π/D

The previous development achieves a significant milestone but only hints at the full expressiveness of fractional types. A small addition that significantly increases expressiveness is to allow ancillae to be initialized to a value of the programmer's choice, not just a globally fixed default value. This ability is used in Grover's search algorithm to implement phase kickback efficiently [40].[3] This can be easily achieved by adapting our language as follows:

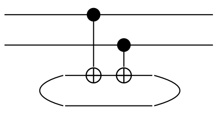
$$
\begin{array}{llll}
\textit{Value types} & t & ::= & \cdots \mid 1/v \\
\textit{Values} & v & ::= & \cdots \mid \circlearrowright \\
\textit{Program types} & & & t \leftrightarrow t \\
\textit{Programs} & c & ::= & \cdots \mid \eta_{v:t} : 1 \leftrightarrow (t \times 1/v) \\
 & & & \mid \epsilon_{v:t} : (t \times 1/v) \leftrightarrow 1
\end{array}
$$

We will call this language Π/D for the fractional extension of Π with a dynamic check. In addition to being more expressive than the language in the previous subsection, Π/D has the advantage of replacing the cumbersome cardinality-based evidence that the type is not empty by the more intuitive evidence of choosing a value of the type. The simplifications are evident in the Agda formalization:

[3]Instead of using an ancilla bit initialized to $|0\rangle$, an ancilla initialized to $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ is used.

```
𝟙/_ : {t : 𝕌} ⟶ ⟦ t ⟧ ⟶ 𝕌

η : {t : 𝕌} (v : ⟦ t ⟧) ⟶ 𝟙 ↔ t ×ᵤ (𝟙/ v)
ε : {t : 𝕌} (v : ⟦ t ⟧) ⟶ t ×ᵤ (𝟙/ v) ↔ 𝟙

interp (η v) tt = just (v , ↻)
interp (ε v) (v’ , ↻) with v ≟ᵤ v’
... | yes _ = just tt
... | no _ = nothing
```

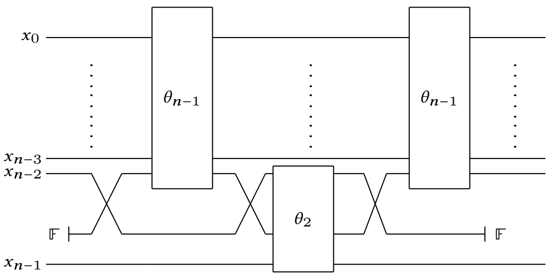
We illustrate the new style of ancilla management with an interesting example used in quantum circuits as a dynamic assertion of entanglement [61]:



```
parity : 𝔹 ×ᵤ 𝔹 ↔ 𝔹 ×ᵤ 𝔹
parity =                 𝔹 ×ᵤ 𝔹
  ↔⟨ uniti₊r ⟩           (𝔹 ×ᵤ 𝔹) ×ᵤ 𝟙
  ↔⟨ id↔ ⊗ (η 𝕋) ⟩       (𝔹 ×ᵤ 𝔹) ×ᵤ (𝔹 ×ᵤ 𝟙/ 𝕋)
  ↔⟨ CNOT13 ⟩            (𝔹 ×ᵤ 𝔹) ×ᵤ (𝔹 ×ᵤ 𝟙/ 𝕋)
  ↔⟨ CNOT23 ⟩            (𝔹 ×ᵤ 𝔹) ×ᵤ (𝔹 ×ᵤ 𝟙/ 𝕋)
  ↔⟨ id↔ ⊗ (ε 𝕋) ⟩       (𝔹 ×ᵤ 𝔹) ×ᵤ 𝟙
  ↔⟨ unite₊r ⟩           (𝔹 ×ᵤ 𝔹) □

t₁ t₂ t₃ t₄ : Maybe ⟦ 𝔹 ×ᵤ 𝔹 ⟧
t₁ = interp parity (𝔽 , 𝔽) -- just (𝔽 , 𝔽)
t₂ = interp parity (𝔽 , 𝕋) -- nothing
t₃ = interp parity (𝕋 , 𝔽) -- nothing
t₄ = interp parity (𝕋 , 𝕋) -- just (𝕋 , 𝕋)
```

The circuit initializes the ancilla to true and checks that it remains true after interacting with the two inputs. If either one of the inputs is true (but not both) the ancilla bit will be negated and the runtime check would fail. Thus the circuit can be used as a runtime assertion that the two inputs are equal. When applied to quantum inputs, the circuit detects whether the two qubits are entangled or not. By varying the initial value of the ancilla, it is possible to customize the dynamic assertion to check for particular entangled states [61].

As a more substantial example, we present a construction (in Fig. 2) of not just the four-bit Toffoli function from Sec. 3.2 but in fact an inductive construction that defines all n-bit Toffoli gates. The construction goes back to Toffoli’s proof of universality [55]:



Note that in the construction of TOFFOLI$_n$ only wire manipulating operators (multiplicative operators), ancilla allocation / de-allocation and TOFFOLI are used.

```
𝔹^_ : ℕ ⟶ 𝕌
𝔹^ zero      = 𝔹
𝔹^ suc n     = (𝔹^ n) ×ᵤ 𝔹

θ : (n : ℕ) ⟶ (𝔹^ n) ↔ (𝔹^ n)
θ = <′-rec (λ n ⟶ (𝔹^ n) ↔ (𝔹^ n)) θ’
  where
    θ’ : (n : ℕ) ⟶ (∀ m ⟶ m <′ n ⟶ (𝔹^ m) ↔ (𝔹^ m)) ⟶ (𝔹^ n) ↔ (𝔹^ n)
    θ’ 0 _ = NOT
    -- CNOT
    θ’ 1 θ” = dist ⨾ (id↔ ⊕ (id↔ ⊗ θ” 0 0<1)) ⨾ factor
    -- TOFFOLI
    θ’ 2 θ” = assocr₊ ⨾ dist ⨾ (id↔ ⊕ (id↔ ⊗ θ” 1 1<2)) ⨾ factor ⨾ assocl₊
    -- TOFFOLIₙ
    θ’ (3+ n) θ” =
      (id↔ ⊗ (uniti₊l ⨾ (η 𝔽 ⊗ id↔) ⨾ assocr₊
               ⨾ (id↔ ⊗ swap₊) ⨾ assocl₊))
      ⨾ assocl₊ ⨾ (assocl₊ ⊗ id↔)
      ⨾ ((θₙ₋₁ ⊗ id↔) ⊗ id↔)
      ⨾ (θ₂ ⊗ id↔)
      ⨾ ((θₙ₋₁ ⊗ id↔) ⊗ id↔)
      ⨾ (assocr₊ ⊗ id↔) ⨾ assocr₊
      ⨾ (id↔ ⊗ (assocr₊ ⨾ (id↔ ⊗ swap₊)
                 ⨾ assocl₊ ⨾ (ε 𝔽 ⊗ id↔) ⨾ unite₊l))
      where
        θₙ₋₁ : (𝔹^ (3 + n)) ↔ (𝔹^ (3 + n))
        θₙ₋₁ = assocr₊ ⨾ (id↔ ⊗ swap₊) ⨾ assocl₊
               ⨾ (θ” (2 + n) 2+n<3+n ⊗ id↔)
               ⨾ assocr₊ ⨾ (id↔ ⊗ swap₊) ⨾ assocl₊

        θ₂ : (𝔹^ (4 + n)) ↔ (𝔹^ (4 + n))
        θ₂ = (assocr₊ ⊗ id↔) ⨾ assocr₊
          ⨾ (id↔ ⊗ θ” 2 2<3+n)
          ⨾ assocl₊ ⨾ (assocl₊ ⊗ id↔)
```

Figure 2. Generalized Toffoli Gates

5 Space-Tracking Abstract Machine

There are two technical reasons we are calling the new type constructor a *fractional type* and hence using for it the suggestive notation $1/t$:

- Categorical: The categorical model for the plain Π language of type isomorphisms is a category with two symmetric monoidal structures (additive and multiplicative) with the multiplicative fragment distributing over the additive one [17]. Generally, a symmetric monoidal category is *compact closed* if every object has a dual object whose behavior is governed by our η and ϵ operators [33]. Our language from the previous section is an adaptation of this idea to the context of Π. Many programming abstractions (such as *name*, *co-name*, and *trace* illustrated in Sec. 6.4) can be derived from properties of compact closed categories. Additionally, all the properties expected of the rational numbers (e.g. $1/(m * n) = 1/m * 1/n$, $1/(1/n) = n$, etc.) can be given constructive content corresponding to manipulation of gc processes. We have already seen the first example and will see more in Sec. 6.4.

- Operational: The other reason our types can be justifiably called fractional is information theoretic. As mentioned in the introduction, the number of bits needed to store information about a space with n distinct values is $\log n$. If the number of bits can be thought of as a debt (to be reclaimed) then a negative number of bits corresponds to a space with $1/n$ distinct values. Indeed we interpret these values as gc processes each specialized to collect space for one of the n distinct values. In this section, we make this intuition precise by presenting a (fully-formalized in Agda) abstract machine that manages an explicit heap.

5.1 Abstract Machine for Π

To start, we introduce some of the common technical details in the simpler setting of the core Π language. The cardinalities of finite types map directly to the natural numbers. An easy proposition to verify is that every combinator must be between types of the same cardinalities; indeed one of the standard interpretations of Π combinators is that they are permutations on finite sets, i.e., they just shuffle elements around without changing the sizes of the underlying sets:

```
card= : {t₁ t₂ : 𝕌} (C : t₁ ↔ t₂) → (∣ t₁ ∣ ≡ ∣ t₂ ∣)
```

To make the space usage explicit in every step of evaluation, we define an abstract machine whose states are pairs of a code register and a memory register. In a low-level implementation the code register would contain a pointer to the current instruction and the memory register would contain a bit-level runtime representation of values with one bit for boolean values, etc. In the Agda formalization, we include the current combinator in the code register, and represent the memory register using an index into an enumeration. Thus a memory register containing a boolean value would be represented as [𝔽 , 𝕋][0] or [𝔽 , 𝕋][1] depending on whether the value is false or true. The prepended vector contains all possible values of the type and the number in between [...] is an index into the vector, representing a finite number that is guaranteed to be in bounds. In a low-level runtime representation only a binary encoding of this finite type would need to be stored, but in order to just prove that space is preserved, we can just inspect the size of the prepended list. Putting it all together, machine states and small-step reductions are formalized as follows:

```
data State' : ℕ → Set where
  ≪_∥_[_]≫ : {A B : 𝕌} →
    A ↔ B → Vec ⟦ A ⟧ ∣ A ∣ → Fin ∣ A ∣ → State' ∣ A ∣
```

```
step' : ∀ {n} → State' n → State' n
```

The constructor for machine states takes a number which is the size of the memory. This size is maintained throughout the execution, even if the intermediate types change. As a small example, the following is a trace of the execution of CNOT applied to 𝕋, 𝕋:

```
≪ dist ⨾ (id↔ ⊕ (id↔ ⊗ swap₊)) ⨾ factor ∥ [(𝔽 , 𝔽) , (𝔽 , 𝕋) , (𝕋 , 𝔽) , (𝕋 , 𝕋)] [ 3 ]≫

≪ id↔ ⨾ (id↔ ⊕ (id↔ ⊗ swap₊)) ⨾ factor ∥ [inj₁ (tt , 𝔽) , inj₁ (tt , 𝕋) , inj₂ (tt , 𝔽) , inj₂ (tt , 𝕋)] [ 3 ]≫

≪ (id↔ ⊕ (id↔ ⊗ swap₊)) ⨾ factor ∥ [inj₁ (tt , 𝔽) , inj₁ (tt , 𝕋) , inj₂ (tt , 𝔽) , inj₂ (tt , 𝕋)] [ 3 ]≫

≪ (id↔ ⊕ (id↔ ⊗ id↔)) ⨾ factor ∥ [inj₁ (tt , 𝔽) , inj₁ (tt , 𝕋) , inj₂ (tt , 𝔽) , inj₂ (tt , 𝕋)] [ 2 ]≫

≪ (id↔ ⊕ id↔) ⨾ factor ∥ [inj₁ (tt , 𝔽) , inj₁ (tt , 𝕋) , inj₂ (tt , 𝔽) , inj₂ (tt , 𝕋)] [ 2 ]≫

≪ id↔ ⨾ factor ∥ [inj₁ (tt , 𝔽) , inj₁ (tt , 𝕋) , inj₂ (tt , 𝔽) , inj₂ (tt , 𝕋)] [ 2 ]≫

≪ factor ∥ [inj₁ (tt , 𝔽) , inj₁ (tt , 𝕋) , inj₂ (tt , 𝔽) , inj₂ (tt , 𝕋)] [ 2 ]≫

≪ id↔ ∥ [(𝔽 , 𝔽) , (𝔽 , 𝕋) , (𝕋 , 𝔽) , (𝕋 , 𝕋)] [ 2 ]≫
```

Note how the size of the vector remains equal to 4 throughout the execution, even if the individual elements change as types change and the current index (i.e., the current value) changes.

5.2 Abstract Machine for Π/D

Using the same assumptions as before, machine states and small step reductions are formalized as follows:

```
data State' : Set where
  ≪_∥_[_]≫ : {A B : 𝕌} →
    A ↔ B → Vec ⟦ A ⟧ ∣ A ∣ → Fin ∣ A ∣ → State'
```

```
step' : State' → State'
```

The main difference is that the state constructor does not take a fixed size and indeed the size changes during execution as appropriate to fit the elements of the type A. In particular, since fractional values have a trivial representation, the cardinality of a fractional type is 1, and hence η used at type A grows the size of the memory by an $|A|$-factor. As an illustrative example, we show a trace of the execution of id' from Sec. 4.3 applied to 𝕋:

```
≪ uniti⁎r ⨾ (id↔ ⊗ η 𝔽) ⨾...∥ [𝔽 , 𝕋] [ 1 ]≫

≪ (id↔ ⊗ η 𝔽) ⨾ ... ∥ [(𝔽 , tt) , (𝕋 , tt)] [ 1 ]≫
⋮
≪ (id↔ ⊗ id↔) ⨾ assocl⁎ ⨾ ... ∥ [(𝔽 , 𝔽 , ↻) , (𝔽 , 𝕋 , ↻) , (𝕋 , 𝔽 , ↻) , (𝕋 , 𝕋 , ↻)] [ 2 ]≫
⋮
≪ assocl⁎ ⨾ ... ∥ [(𝔽 , 𝔽 , ↻) , (𝔽 , 𝕋 , ↻) , (𝕋 , 𝔽 , ↻) , (𝕋 , 𝕋 , ↻)] [ 2 ]≫
⋮
≪ (((dist ⨾ ...) ⊗ id↔) ⨾ ...∥ [((𝔽 , 𝔽) , ↻) , ((𝔽 , 𝕋) , ↻) , ((𝕋 , 𝔽) , ↻) , ((𝕋 , 𝕋) , ↻)] [ 2 ]≫
⋮
≪((((id↔ ⊕ (id↔ ⊗ swap₊)) ⨾ ...) ⊗ id↔) ⨾ ...∥ [(inj₁ (tt , 𝔽) , ↻) , (inj₁ (tt , 𝕋) , ↻) , (inj₂ (tt , 𝔽) , ↻) , (inj₂ (tt , 𝕋) , ↻)] [ 2 ]≫
⋮
≪((factor ⨾...) ⊗ id↔)⨾...∥ [(inj₁ (tt , 𝔽) , ↻) , (inj₁ (tt , 𝕋) , ↻) , (inj₂ (tt , 𝔽) , ↻) , (inj₂ (tt , 𝕋) , ↻)] [ 3 ]≫
⋮
≪(((distl ⨾ ...) ⊗ id↔) ⨾...∥ [((𝔽 , 𝔽) , ↻) , ((𝔽 , 𝕋) , ↻) , ((𝕋 , 𝔽) , ↻) , ((𝕋 , 𝕋) , ↻)] [ 3 ]≫
⋮
≪((((id↔ ⊕ (swap₊ ⊗ id↔)) ⨾ ...) ⊗ id↔) ⨾ ... ∥ [(inj₁ (𝔽 , tt) , ↻) , (inj₁ (𝕋 , tt) , ↻) , (inj₂ (𝔽 , tt) , ↻) , (inj₂ (𝕋 , tt) , ↻)] [ 3 ]≫
⋮
≪ ((factorl ⨾ swap⁎) ⊗ id↔) ⨾ ... ∥ [(inj₁ (𝔽 , tt) , ↻) , (inj₁ (𝕋 , tt) , ↻) , (inj₂ (𝔽 , tt) , ↻) , (inj₂ (𝕋 , tt) , ↻)] [ 2 ]≫
⋮
≪ (swap⁎ ⊗ id↔) ⨾ assocr⁎ ⨾ ... ∥ [((𝔽 , 𝔽) , ↻) , ((𝔽 , 𝕋) , ↻) , ((𝕋 , 𝔽) , ↻) , ((𝕋 , 𝕋) , ↻)] [ 1 ]≫
⋮
≪ assocr⁎ ⨾ (id↔ ⊗ ε 𝔽) ⨾ unite⁎r ∥ [((𝔽 , 𝔽) , ↻) , ((𝔽 , 𝕋) , ↻) , ((𝕋 , 𝔽) , ↻) , ((𝕋 , 𝕋) , ↻)] [ 2 ]≫
⋮
≪ (id↔ ⊗ ε 𝔽) ⨾ unite⁎r ∥ [(𝔽 , 𝔽 , ↻) , (𝔽 , 𝕋 , ↻) , (𝕋 , 𝔽 , ↻) , (𝕋 , 𝕋 , ↻)] [ 2 ]≫
⋮
≪ unite⁎r ∥ [(𝔽 , tt) , (𝕋 , tt)] [ 1 ]≫

≪ id↔ ∥ [𝔽 , 𝕋] [ 1 ]≫
```

6 Dependently-Typed Garbage Collectors

By lifting the scoping restriction, the development in the previous sections is already more general than the state of the art in ancilla management. It however shares the same limitation of requiring a runtime check to ensure ancillae values are properly restored to their allocation value [26, 54]. In this section, we address this limitation using a combination of pointed types, singleton types, monads, and comonads.

6.1 Lifting Evaluation to the Type System

Before giving all the (rather involved) technical details, we highlight the main idea using the toy language below:

```
data T : Set where
  N : T
  B : T

⟦_⟧ : T ⟶ Set
⟦ N ⟧ = ℕ
⟦ B ⟧ = Bool

data Fun : T ⟶ T ⟶ Set where
  square   : Fun N N
  isZero   : Fun N B
  compose : {a b c : T} ⟶ Fun b c ⟶ Fun a b ⟶ Fun a c

eval : {a b : T} ⟶ Fun a b ⟶ ⟦ a ⟧ ⟶ ⟦ b ⟧
eval square n = n * n
eval isZero 0 = true
eval isZero (suc _) = false
eval (compose g f) v = eval g (eval f v)
```

The toy language has two types (natural numbers and booleans) and two functions (and their compositions). Say we wanted to prove that compose isZero square always returns false when applied to a non-zero natural number. We can certainly do this proof in Agda (i.e., in the meta-language of our formalization) but we would like to do the proof within the toy language itself. The most important reason is that it can then be used within the language to optimize programs (or, for the case of Π/D to remove a runtime check).

The strategy we adopt is to create a lifted version of the toy language with *pointed types* [53], i.e., types paired with a value of the type. In the lifted language, the evaluation function has an interesting type: it keeps track of the result of evaluation within the type:

```
data T· : Set where
  _#_ : (a : T) ⟶ (v : ⟦ a ⟧) ⟶ T·

⟦_⟧· : T· ⟶ Σ[ A ∈ Set ] A
⟦ T # v ⟧· = ⟦ T ⟧ , v

data Fun· : T· ⟶ T· ⟶ Set where
  lift : {a b : T} {v : ⟦ a ⟧} ⟶ (f : Fun a b) ⟶
         Fun· (a # v) (b # (eval f v))
```

Various properties of compose isZero square can be derived within the extended type system:

```
test1 : Fun· (N # 3) (B # false)
test1 = lift (compose isZero square)

test2 : Fun· (N # 0) (B # true)
test2 = lift (compose isZero square)

test3 : ∀ {n} ⟶ Fun· (N # (suc n)) (B # false)
test3 = lift (compose isZero square)
```

We can track concrete values as the first two tests show. More interesting is test3 which shows a property that holds for all natural numbers n. Evaluation proceeds "symbolically" in the type system. From the definition of eval, it is clear that eval square (suc n) produces (suc n) * (suc n) which by definition of multiplication is an expression with a leading suc constructor which is enough to determine that evaluating isZero on it yields false. This form of partial evaluation will prove to be quite expressive, allowing us to keep track of ancillae values throughout complex programs.

6.2 Pointed and Singleton Types: $\Pi/\bullet$

Proceeding similarly, we create a version of Π with pointed types which we call $\Pi/\bullet$. In addition to the pointed types of the form $t\#v$, we need a multiplicative structure over pointed types, fractional types, and a special kind of pointed type that includes just one value, a singleton type. The singleton types will allow the type system to track the flow of one particular value, which is exactly what is needed to prove the safety of de-allocation:

```
Singleton : (A : Set) ⟶ (v : A) ⟶ Set
Singleton A v = ∃ (λ ● ⟶ v ≡ ●)

Recip : (A : Set) ⟶ (v : A) ⟶ Set
Recip A v = Singleton A v ⟶ ⊤

data ·𝕌 : Set where
  _#_     : (t : 𝕌) ⟶ (v : ⟦ t ⟧) ⟶ ·𝕌
  _·×ᵤ_ : ·𝕌 ⟶ ·𝕌 ⟶ ·𝕌
  ⦇_⦈     : ·𝕌 ⟶ ·𝕌
  ·𝟙/     : ·𝕌 ⟶ ·𝕌

·⟦_⟧ : ·𝕌 ⟶ Σ[ A ∈ Set ] A
·⟦ t # v ⟧         = ⟦ t ⟧ , v
·⟦ T₁ ·×ᵤ T₂ ⟧ = let (t₁ , v₁) = ·⟦ T₁ ⟧
                          (t₂ , v₂) = ·⟦ T₂ ⟧
                      in  (t₁ × t₂) , (v₁ , v₂)
·⟦ ⦇ T ⦈ ⟧       = let (t , v) = ·⟦ T ⟧ in Singleton t v , (v , refl)
·⟦ ·𝟙/ T ⟧       = let (t , v) = ·⟦ T ⟧ in Recip t v , λ _ ⟶ tt
```

As can be seen, a lifted type is interpreted as a type along with a corresponding value. A singleton type consists a special value of a given type: any claim that a value belongs to the singleton type must come with a proof that this value is equivalent to the special value. A fractional type consumes a particular singleton type returning the unit type. Note that the runtime gc process simply disregards the value it is given: the type system however prevents this value from being applied to anything but the particular singleton value in question.

The combinators in the lifted language $\Pi/\bullet$ consist of all the combinators in the core Π language but now operating on pointed types. The mediation between general pointed types and singleton types is done via return and extract, which as we prove below form a dual monad/comonad pair. The types for η and ϵ are now specialized to guarantee safety of de-allocation as follows. When applying η at a pointed type, the current witness value is put in focus in a singleton type and a gc process for that particular singleton type is created. To apply this process using ϵ the very same singleton value must be the current one.

```
data _·⟶_ : ·𝕌 → ·𝕌 → Set where
  -- lifting from plain Π
  ·c      : {t1 t2 : 𝕌} {v : ⟦ t1 ⟧} →
            (c : t1 ↔ t2) → t1 # v ·⟶ t2 # (eval c v)
  ·times# : {t1 t2 : 𝕌} {v1 : ⟦ t1 ⟧} {v2 : ⟦ t2 ⟧} →
            ((t1 ×u t2) # (v1 , v2)) ·⟶ ((t1 # v1) ·×u (t2 # v2))
  ·#times : {t1 t2 : 𝕌} {v1 : ⟦ t1 ⟧} {v2 : ⟦ t2 ⟧} →
            ((t1 # v1) ·×u (t2 # v2)) ·⟶ ((t1 ×u t2) # (v1 , v2))
  -- multiplicative structure (omitted)
  -- monad / comonad
  return  : (T : ·𝕌) → T ·⟶ ⦇ T ⦈
  extract : (T : ·𝕌) → ⦇ T ⦈ ·⟶ T
  -- eta/epsilon
  η : (T : ·𝕌) → ·𝟙 ·⟶ ⦇ T ⦈ ·×u ·𝟙/ T
  ε : (T : ·𝕌) → ⦇ T ⦈ ·×u ·𝟙/ T ·⟶ ·𝟙
```

The mediation between pointed and singleton types is based on a dual monad/comonad pair from which many structural properties can be derived.

Proposition 6.1. *⦇·⦈ is both an idempotent strong monad and an idempotent costrong comonad over pointed types.*

Proof. The main insight needed is to define the functor $\cdot\mathsf{Sing}_u$, the tensor/cotensor, and the join/cojoin (duplicate):

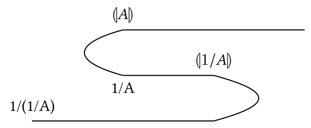
```
·Singu : {T1 T2 : ·𝕌} → (T1 ·⟶ T2) → ⦇ T1 ⦈ ·⟶ ⦇ T2 ⦈
·Singu {T1} {T2} c = extract T1 ·⨟ c ·⨟ return T2

tensor : {T1 T2 : ·𝕌} → ⦇ T1 ⦈ ·×u ⦇ T2 ⦈ ·⟶ ⦇ T1 ·×u T2 ⦈
tensor {T1} {T2} = (extract T1 ·⊗ extract T2) ·⨟ return (T1 ·×u T2)

cotensor : {T1 T2 : ·𝕌} → ⦇ T1 ·×u T2 ⦈ ·⟶ ⦇ T1 ⦈ ·×u ⦇ T2 ⦈
cotensor {T1} {T2} = extract (T1 ·×u T2) ·⨟ (return T1 ·⊗ return T2)

join : {T1 : ·𝕌} → ⦇ ⦇ T1 ⦈ ⦈ ·⟶ ⦇ T1 ⦈
join {T1} = extract ⦇ T1 ⦈

duplicate : {T1 : ·𝕌} → ⦇ T1 ⦈ ·⟶ ⦇ ⦇ T1 ⦈ ⦈
duplicate {T1} = return ⦇ T1 ⦈
```

□

Like for the toy language, evaluation is reflected in the type system, and in this case we have the additional property that evaluation is reversible:

```
·eval : {T1 T2 : ·𝕌} → (C : T1 ·⟶ T2) →
  let (t1 , v1) = ·⟦ T1 ⟧; (t2 , v2) = ·⟦ T2 ⟧
  in Σ (t1 → t2) (λ f → f v1 ≡ v2)

!·_ : {A B : ·𝕌} → A ·⟶ B → B ·⟶ A
```

Thus, to summarize, if a combinator expects a singleton type, then it would only typecheck in the lifted language, if it is given the unique value it expects. A particularly intriguing instance of that situation is the following program:



```
revrev : {A : ·𝕌} → ·𝟙/ (·𝟙/ A) ·⟶ ⦇ A ⦈
revrev {A} =
  ·uniti⋆l ·⨟
  (η A ·⊗ ·id↔) ·⨟
  ((·id↔ ·⊗ return (·𝟙/ A)) ·⊗ ·id↔) ·⨟
  ·assocr⋆ ·⨟
  ·id↔ ·⊗ ε (·𝟙/ A) ·⨟
  ·unite⋆r
```

The program takes a value of type $\cdot\mathbb{1}/\ (\cdot\mathbb{1}/\ A)$. This would be a gc process specialized to collect another gc process! At runtime, there would be no information other than the functions that ignore their argument but the type system provides enough guarantees to ensure that this process is well-defined and safe. The following trace shows the actual execution for confirmation:

```
≪ uniti⋆l ⨟ (η 𝕋 ⊗ id↔) ⨟ ... ∥ [↻] [ 0 ]≫
⋮
≪ (η 𝕋 ⊗ id↔) ⨟ assocr⋆ ⨟ ... ∥ [(tt , ↻)] [ 0 ]≫
⋮
≪ assocr⋆ ⨟ (id↔ ⊗ ε ↻) ⨟ unite⋆r ∥ [((𝔽 , ↻) , ↻) , ((𝕋 , ↻) , ↻)] [ 1 ]≫
⋮
≪ (id↔ ⊗ ε ↻) ⨟ unite⋆r ∥ [(𝔽 , ↻ , ↻) , (𝕋 , ↻ , ↻)] [ 1 ]≫
⋮
≪ unite⋆r ∥ [(𝔽 , tt) , (𝕋 , tt)] [ 1 ]≫

≪ id↔ ∥ [𝔽 , 𝕋] [1 ]≫
```

As the trace shows, by garbage collecting the collector for value 𝕋, this value which was maintained in the type is now "rematerialized."

6.3 Extraction

By lifting programs and their evaluation to the type level, we can naturally leverage the typechecking process to verify properties of interest, including the safe de-allocation of ancillae. One "could" just forget about Π/D and instead use $\Pi/\bullet$ as *the* programming language for ancilla management. Indeed the dual nature of proofs and programs is more and more exploited in languages like the one used to formalize this paper (Agda). The examples in Sec. 6.4 are written in that style and they are relatively similar to programs in the "normal" Π/D language.

However, it is also often the case than constructive proofs are further processed to extract native efficient programs that eschew the overhead of maintaining information needed just for proof invariants. In our case, the question is whether we can extract from a $\Pi/\bullet$ program, a program in Π/D that uses a simpler type system, a simpler runtime representation, and yet is guaranteed to be safe and hence can run without the runtime checks associated with de-allocation sites. In this section, we show that this indeed the case.

The extraction map is fully implemented in the underlying Agda formalization. We present the most significant highlights. Here are three important functions' signatures:

```
Ext𝕌    : ·𝕌 → Σ[ t ∈ 𝕌D ] ⟦ t ⟧

Ext·⟶ : ∀ {t_1 t_2} → t_1 ·⟶ t_2 →
              proj_1 (Ext𝕌 t_1) ↔ proj_1 (Ext𝕌 t_2)

Ext≡ : ∀ {t_1 t_2} → (c : t_1 ·⟶ t_2)
     → let (t_1' , v_1') = Ext𝕌 t_1
              (t_2' , v_2') = Ext𝕌 t_2
         in interp (Ext·⟶ c) v_1' ≡ just v_2'
```

The function Ext𝕌 maps a $\Pi/\bullet$ type to a Π/D type and a value in the type. The function Ext·⟶ maps a $\Pi/\bullet$ combinator to a Π/D combinator. And finally the function Ext≡ asserts that extracted code cannot throw an exception (it must return just...).

Each of these functions has one or two enlightening cases which we explain.

In Π/D the fractional type expresses that it expects a particular value but lacks any mechanisms to enforce this requirement. Thus we have no choice when mapping a fractional type from $\Pi/\bullet$ to Π/D than to just use this 𝟙/ v type:

```
Ext𝕌 (·𝟙/ T) with Ext𝕌 T
... | (t , v) = 𝟙/ v , ↻
```

When mapping $\Pi/\bullet$ combinators to Π/D combinators, the main interesting cases are for η and ε. In those, we just use the values from the pointed type as choices for the ancilla value and expectation for the gc process:

```
Ext·⟶ (η T) = η (proj_2 (Ext𝕌 T))
Ext·⟶ (ε T) = ε (proj_2 (Ext𝕌 T))
```

Finally we can prove the correctness of the extraction. The punchline is in the following case:

```
Ext≡ (ε T) | yes p = refl
Ext≡ (ε T) | no ¬p = ⊥-elim (¬p refl)
```

In this case, the singleton type in $\Pi/\bullet$ guarantees that the runtime check cannot fail.

6.4 Examples

In addition to the examples presented throughout the paper, we present examples that provide novel perspectives on gc processes, first as space transformers, and second as tracing operators. Both ideas are inspired from known constructions in compact closed categories [2].

Space Transformers. A pair of a value and a gc process cancel each other when the gc process is expecting this particular value. In more general cases, the pair can be viewed as a space transformer that represents an implicit function that converts the value to the one expected by the gc process. Such space transformers can be composed and applied.

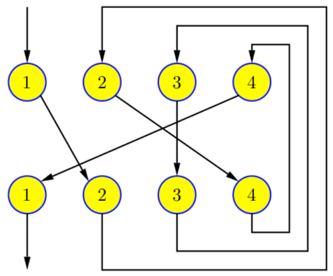
```
_⊸_ : (A B : ·𝕌) → ·𝕌
A ⊸ B = ·𝟙/ A ·×_u ⦇ B ⦈

id⊸ : {A : ·𝕌} → (A ⊸ A) ·⟶ ·𝟙
id⊸ {A} = ·swap_* ·⨾ ε A

name : {A B : ·𝕌} → (f : A ·⟶ B) → ·𝟙 ·⟶ (A ⊸ B)
name {A} {B} f = η A ·⨾ (·Sing_u f ·⊗ ·id↔) ·⨾ ·swap_*

coname : {A B : ·𝕌} → (f : A ·⟶ B) → (B ⊸ A) ·⟶ ·𝟙
coname {A} {B} f = ·swap_* ·⨾ (·Sing_u f ·⊗ ·id↔) ·⨾ ε B

comp⊸ : (A B C : ·𝕌) → (A ⊸ B) ·×_u (B ⊸ C) ·⟶ (A ⊸ C)
app       : (A B : ·𝕌) → (A ⊸ B) ·×_u ⦇ A ⦈ ·⟶ ⦇ B ⦈
dist×/    : {A B C D : ·𝕌} →
              (A ⊸ B) ·×_u (C ⊸ D) ·⟶ ((A ·×_u C) ⊸ (B ·×_u D))
```

Tracing Operators. Another intriguing abstraction is based on the fact that dual objects can be used to define a tracing operator:

```
trace : {A B C : ·𝕌} → (f : A ·×_u C ·⟶ B ·×_u C) → A ·⟶ B
trace {A} {B} {C} f =
  ·uniti_*r ·⨾ (return _ ·⊗ η C) ·⨾ ·assocl_* ·⨾
  (tensor ·⊗ ·id↔) ·⨾
  (·Sing_u f ·⊗ ·id↔) ·⨾
  (cotensor ·⊗ ·id↔) ·⨾
  ·assocr_* ·⨾ (extract _ ·⊗ ε C) ·⨾ ·unite_*r
```

The operator takes a function that expects a pair of inputs A and C and returns a pair of outputs B and C. In the world of reversible programs (or permutations), the larger permutation guarantees the existence of a smaller permutation that permutes A to B. Indeed, we can programmatically construct such a permutation:

As the diagram below (taken from Abramsky's paper [2]) illustrates, this is not a trivial construction in the sense that the permutation from A to B uses C in non-trivial ways.



Interpreted in our setting, the program can be written as follows:

```
traceA : {A₁ A₂ A₃ A₄ : ·𝕌}
  → (f₁ : A₁ ·⟶ A₂) → (f₂ : A₂ ·⟶ A₄)
  → (f₃ : A₃ ·⟶ A₃) → (f₄ : A₄ ·⟶ A₁)
  → A₁ ·⟶ A₁
traceA f₁ f₂ f₃ f₄ = trace f
  where f = (f₁ ·⊗ (f₂ ·⊗ (f₃ ·⊗ f₄))) ·⨟
                ·assocl⋆ ·⨟ ·swap⋆ ·⨟ ·swap⋆ ·⊗ ·id↔ ·⨟
                ·assocr⋆ ·⨟ (·id↔ ·⊗ ·assocl⋆) ·⨟
                ·id↔ ·⊗ (·swap⋆ ·⊗ ·id↔ ·⨟ ·assocr⋆)
```

The feedback loop in the diagram is realized using ancillae allocation and de-allocation. The safety condition on de-allocation ensures that the allocated value (now in the feedback loop) matches the de-allocated value. Thus, the entire traced component is treated as an ancilla value used internally to perform the larger program but leaving no memory footprint.

7 Related Work and Conclusion

We have introduced, in the context of reversible/quantum languages, the concept of fractional types as descriptions of specialized gc processes. Although the basic idea is rather simple and intuitive, the technical details needed to reason about individual values are somewhat intricate.

There has been ample work on ancillae before us. For example [43] explicitly represents the lifetime of quantum wires, enabling quantum wires to be recycled, leading to more efficient circuits. Interesting applications of ancillae is in the recent paper [61] on quantum runtime assertions. In section 4.3, we already mentioned dynamic assertion of entanglement; they also show how to provide (dynamic) assertions on classical states, and testing of superposition states. Our techniques should combine with theirs to provide even more possibilities for dynamic assertions. Ancilla management is a complex business, and the verified circuit compiler ReVerC [8] goes to great pains to ensure that ancilla are dealt with properly. Both QCL [42] and Quipper [26] also do automatic (but not certified) ancilla management during code generation. This remains an active area of research, as outlined in the CACM paper [56].

Another interesting avenue is to understand how to merge Π/● with other languages for quantum computing [21, 50, 57]. One particular language, QWIRE [46], has a paradigm that appears eminently compatible to ours and has many desirable meta-properties that have been formally proven to hold. QWIRE however has an unguarded `discard` directive that may lead to erroneous circuits. For example, their circuit reversing operation is partial, whereas ours is provably total.

When looking at higher-level reversible languages, Janus comes to mind [38]. More recently, [60] shows how to do clean, i.e. *garbage free* computation. It would certainly be intriguing to augment Janus with a certified garbage collection discipline such as ours, so that one might hope that the resulting computations might be more efficiently done.

Another different thread of work stems from categorical models of quantum computation, especially those based on dagger compact closed categories as introduced in [5], and greatly expanded upon in [6, 48, 49], or more recently in [29]. Various results can be obtained in that general setting, such as the no-cloning theorem [4]. The 2016 book [18] is a good survey of the developments since. Of particular note for our applications are two 2005 papers by Abramsky [2, 3] with explicit examples of circuits using trace that are very useful tests for our approach.

References

[1] Scott Aaronson, Daniel Grier, and Luke Schaeffer. 2017. The Classification of Reversible Bit Operations. In *8th Innovations in Theoretical Computer Science Conference (ITCS 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Christos H. Papadimitriou (Ed.), Vol. 67. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 23:1–23:34. https://doi.org/10.4230/LIPIcs.ITCS.2017.23

[2] Samson Abramsky. 2005. Abstract Scalars, Loops, and Free Traced and Strongly Compact Closed Categories. In *CALCO*.

[3] Samson Abramsky. 2005. A structural approach to reversible computation. *Theor. Comput. Sci.* 347 (December 2005), 441–464. Issue 3.

[4] Samson Abramsky. 2009. No-cloning in categorical quantum mechanics. *arXiv preprint arXiv:0910.2401* (2009).

[5] Samson Abramsky and Bob Coecke. 2004. A Categorical Semantics of Quantum Protocols. In *LICS*.

[6] Samson Abramsky and Bob Coecke. 2009. Categorical quantum mechanics. *Handbook of quantum logic and quantum structures* 2 (2009), 261–325.

[7] Dorit Aharonov. 2003. A Simple Proof that Toffoli and Hadamard are Quantum Universal. (2003). arXiv:quant-ph/0301040.

[8] Matthew Amy, Martin Roetteler, and Krysta M Svore. 2017. Verified compilation of space-efficient reversible circuits. In *International Conference on Computer Aided Verification*. Springer, 3–21.

[9] Jean Bénabou. 1963. Catégories avec multiplication. *C. R. de l'Académie des Sciences de Paris* 256, 9 (1963), 1887–1890.

[10] Jean Bénabou. 1964. Algèbre élémentaire dans les catégories avec multiplication. *C. R. Acad. Sci. Paris* 258, 9 (1964), 771–774.

[11] C.H. Bennett. 2003. Notes on Landauer's principle, reversible computation, and Maxwell's Demon. *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics* 34, 3 (2003), 501–510.

[12] C.H. Bennett. 2010. Notes on the history of reversible computation. *IBM Journal of Research and Development* 32, 1 (2010), 16–23.

[13] C.H. Bennett and R. Landauer. 1985. The fundamental physical limits of computation. *Scientific American* 253, 1 (1985), 48–56.
[14] C. H. Bennett. 1973. Logical reversibility of computation. *IBM J. Res. Dev.* 17 (November 1973), 525–532. Issue 6.
[15] P Nick Benton. 1994. A mixed linear and non-linear logic: Proofs, terms and models. In *International Workshop on Computer Science Logic*. Springer, 121–135.
[16] William J. Bowman, Roshan P. James, and Amr Sabry. 2011. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *RC*.
[17] Jacques Carette and Amr Sabry. 2016. Computing with Semirings and Weak Rig Groupoids. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9632. Springer, 123–148.
[18] Giulio Chiribella and Robert W Spekkens. 2016. *Quantum theory: Informational foundations and foils*. Springer.
[19] Ronald de Wolf. 2019. Quantum Computing: Lecture Notes. (2019). https://homepages.cwi.nl/~rdewolf/qcnotesv2.pdf.
[20] DGBJ Dieks. 1982. Communication by EPR devices. *Physics Letters A* 92, 6 (1982), 271–272.
[21] Gilles Dowek and Pablo Arrighi. 2017. Lineal: A linear-algebraic Lambda-calculus. *Logical Methods in Computer Science* 13 (2017).
[22] Marcelo Fiore. 2004. Isomorphisms of generic recursive polynomial types. In *POPL*. ACM, 77–88.
[23] M. P. Fiore, R. Di Cosmo, and V. Balat. 2006. Remarks on Isomorphisms in Typed Calculi with Empty and Sum Types. *Annals of Pure and Applied Logic* 141, 1-2 (2006), 35–50.
[24] Yongshan Ding Fred Chong, Ken Brown. 2019. The Case for Quantum Computing. (2019). https://www.sigarch.org/the-case-for-quantum-computing/.
[25] E. Fredkin and T. Toffoli. 1982. Conservative logic. *International Journal of Theoretical Physics* 21, 3 (1982), 219–253.
[26] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 333–342. https://doi.org/10.1145/2491956.2462177
[27] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing (STOC '96)*. ACM, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866
[28] Yipeng Huang and Margaret Martonosi. 2018. QDB: From Quantum Algorithms Towards Correct Quantum Programs. In *9th Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU@SPLASH 2018, November 5, 2018, Boston, Massachusetts, USA*. 4:1–4:14. https://doi.org/10.4230/OASIcs.PLATEAU.2018.4
[29] Mathieu Huot and Sam Staton. 2019. Quantum channels as a categorical completion. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. 1–13. https://doi.org/10.1109/LICS.2019.8785700
[30] Roshan P. James and Amr Sabry. 2012. Information effects. In *POPL*. ACM, 73–84.
[31] Roshan P. James and Amr Sabry. 2012. Isomorphic Interpreters from Logically Reversible Abstract Machines. In *RC*.
[32] Roshan P James and Amr Sabry. 2014. Theseus: A high level language for reversible computing. In *Work-in-progress report in the Conference on Reversible Computation*.
[33] G Maxwell Kelly. 1972. Many-variable functorial calculus. I. In *Coherence in categories*. Springer, 66–105.
[34] Neelakantan R Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating dependent and linear types. *POPL 2015* (2015).
[35] Rolf Landauer. 1961. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.* 5 (July 1961), 183–191. Issue 3.
[36] Rolf Landauer. 1996. The physical nature of information. *Physics Letters A* (1996).
[37] Miguel L. Laplaza. 1972. Coherence for distributivity. In *Coherence in Categories*, G.M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane (Eds.). Lecture Notes in Mathematics, Vol. 281. Springer Verlag, Berlin, 29–65. https://doi.org/10.1007/BFb0059555
[38] C. Lutz and H. Derby. 1982. Janus: a time-reversible language. *Unpublished report, Ph. D. Dissertation, California Institute of Technology* (1982).
[39] Saunders MacLane. 1963. Natural associativity and commutativity. *Rice Institute Pamphlet-Rice University Studies* 49, 4 (1963).
[40] Andy Matuschak and Michael A. Nielsen. 2019. How does the quantum search algorithm work? (2019). https://quantum.country/search.
[41] National Academies of Sciences, Engineering, and Medicine. 2019. *Quantum Computing: Progress and Prospects*. The National Academies Press, Washington, DC. https://doi.org/10.17226/25196
[42] Bernhard Ömer. 2002. Procedural quantum programming. In *AIP Conference Proceedings*, Vol. 627. AIP, 276–285.
[43] Alexandru Paler, Robert Wille, and Simon J. Devitt. 2016. Wire recycling for quantum circuit optimization. *Phys. Rev. A* 94 (Oct 2016), 042337. Issue 4. https://doi.org/10.1103/PhysRevA.94.042337
[44] P. Panangaden and É.O. Paquette. 2011. *A Categorical Presentation of Quantum Computation with Anyons*. Springer Berlin Heidelberg, Berlin, Heidelberg, 983–1025. https://doi.org/10.1007/978-3-642-12821-9_15
[45] James L Park. 1970. The concept of transition in quantum mechanics. *Foundations of Physics* 1, 1 (1970), 23–33.
[46] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 846–858. https://doi.org/10.1145/3009837.3009894
[47] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. 2018. From Symmetric Pattern-Matching to Quantum Control. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 348–364.
[48] P. Selinger. 2007. Dagger compact closed categories and completely positive maps. *Electronic Notes in Theoretical Computer Science* 170 (2007), 139–163.
[49] Peter Selinger. 2011. Finite Dimensional Hilbert Spaces are Complete for Dagger Compact Closed Categories (Extended Abstract). *Electron. Notes Theor. Comput. Sci.* 270, 1 (2011).
[50] Peter Selinger and Benoît Valiron. 2006. A lambda calculus for quantum computation with classical control. *MSCS* 16, 3 (2006), 527–552.
[51] Claude Elwood Shannon. 1948. A mathematical theory of communication. *Bell Systems Technical Journal* 27 (1948), 379–423,623–656.
[52] Zachary Sparks and Amr Sabry. 2014. Superstructural Reversible Logic. In *3rd International Workshop on Linearity*.
[53] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study.
[54] Michael Kirkedal Thomsen, Robin Kaarsgaard, and Mathias Soeken. 2015. Ricercar: A Language for Describing and Rewriting Reversible Circuits with Ancillae and Its Permutation Semantics. In *Reversible Computation*, Jean Krivine and Jean-Bernard Stefani (Eds.). Springer International Publishing, Cham, 200–215.
[55] Tommaso Toffoli. 1980. Reversible Computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Springer-Verlag, 632–644.
[56] Benoît Valiron, Neil J Ross, Peter Selinger, D Scott Alexander, and Jonathan M Smith. 2015. Programming the quantum future. *Commun. ACM* 58, 8 (2015), 52–61.
[57] A. van Tonder. 2004. A Lambda Calculus for Quantum Computation. *SIAM J. Comput.* 33, 5 (2004), 1109–1135.

[58] William K Wootters and Wojciech H Zurek. 1982. A single quantum cannot be cloned. *Nature* 299, 5886 (1982), 802.

[59] Siyao Xu. 2015. Reversible Logic Synthesis with Minimal Usage of Ancilla Bits. *CoRR* abs/1506.03777 (2015). arXiv:1506.03777 http://arxiv.org/abs/1506.03777

[60] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2008. Principles of a reversible programming language. In *Conference on Computing Frontiers*. ACM, 43–54.

[61] Huiyang Zhou and Gregory T. Byrd. 2019. Quantum Circuits for Dynamic Runtime Assertions in Quantum Computation. *Computer Architecture Letters* 18, 2 (2019), 111–114. https://doi.org/10.1109/LCA.2019.2935049