Embracing the Laws of Physics: Three Reversible Models of Computation

Jacques Carette
McMaster University

Roshan P. James
Google

Amr Sabry
Indiana University

Abstract

Our main models of computation (the Turing Machine and the RAM) and most modern computer architectures make fundamental assumptions about which primitive operations are realizable on a physical computing device. The consensus is that these primitive operations include logical operations like conjunction, disjunction and negation, as well as reading and writing to a large collection of memory locations. This perspective conforms to a macro-level view of physics and indeed these operations are realizable using macro-level devices involving thousands of electrons. This point of view is however incompatible with computation realized using quantum devices or analyzed using elementary thermodynamics as both these fundamental physical theories imply that information is a conserved quantity of physical processes and hence of primitive computational operations.

Our aim is to re-develop foundational computational models in a way that embraces the principle of conservation of information. We first define what information is and what its conservation means in a computational setting. We emphasize the idea that computations must be reversible transformations on data. One can think of data as modeled using topological spaces and programs as modeled by reversible deformations of these spaces. We then illustrate this idea using three notions of data and their associated reversible computational models. The first instance only assumes unstructured finite data, i.e., discrete topological spaces. The corresponding notion of reversible computation is that of permutations. We show how this simple model subsumes conventional computations on finite sets. We then consider a modern structured notion of data based on the Curry-Howard correspondence between logic and type theory. We develop the corresponding notion of reversible deformations using a sound and complete programming language for witnessing type isomorphisms and proof terms for commutative semirings. We then "move up a level" to examine spaces that treat programs as data, which is a crucial notion for any universal model of computation. To derive the corresponding notion of reversible programs between programs, i.e., reversible program equivalences, we look at the "higher dimensional" analog to commutative semirings: symmetric rig groupoids. The coherence laws for these groupoids turn out to be exactly the sound and complete reversible program equivalences we seek.

We conclude with some possible generalizations inspired by homotopy type theory and survey several open directions for further research.

1 Reversibility, the Missing Principle

What kind of operations can computers perform? This question has been answered several times in the last hundred years, where each answer proposes an abstract *model of computation* that specifies allowable operations and (usually) their cost. The emerging consensus, reflected in both early models of computations such as the Turing Machine and the RAM as well as in the early Von Neumann models and in modern computer architectures, is that basic computer operations include logical operations like conjunction, disjunction, and negation, as well as reading from and writing to a large (infinite) collection of memory locations. From this small set of primitive operations emerges all higher-level programming languages and abstractions.

No doubt, this consensus on the available primitive physical operations has been successful. Futhermore, these operations *can* indeed be performed on a computer. Yet, today, with a possible quantum computing revolution in sight and an unprecedented explosion in embedded computers and cyber-physical systems, there

are reasons to re-think this foundational question. In fact, the calls to re-think this foundational question have been proclaimed by physicists almost forty years ago:

> **Toffoli 1980 [1]:** Mathematical models of computation are abstract constructions, by their nature unfettered by physical laws. However, if these models are to give indications that are relevant to concrete computing, they must somehow capture, albeit in a selective and stylized way, certain general physical restrictions to which all concrete computing processes are subjected.
>
> **Feynman 1982 [2]:** Another thing that has been suggested early was that natural laws are reversible, but that computer rules are not. But this turned out to be false; the computer rules can be reversible, and it has been a very, very useful thing to notice and to discover that. This is a place where the relationship of physics and computation has turned itself the other way and told us something about the possibilities of computation. So this is an interesting subject because it tells us something about computer rules.

These quotes by Toffoli and Feynman both highlight the consequences of two obvious observations: (i) all the operations that a computer performs reduce to basic physical operations; and (ii) there is a mismatch between the logical operations of a typical model of computation (which are logically irreversible) and the fundamental laws of physics (which are reversible). One could certainly dismiss the mismatch as irrelevant to the practice of computing but our thesis is that the next computing revolution is likely to be founded on revised models of computation that are designed to be in closer harmony with the laws of physics.

After a detailed introduction on the origins of *logically reversibile computer operations* and an excursion into the origins of *irreversible computer operations*, we will develop in detail three reversible models of computation and discuss their potential applications.

Maxwell's Daemon. To fully appreciate the missing principle of *reversibility* in conventional computing, we go back to an old thought experiment by J. C. Maxwell. The details are codified in a letter that Maxwell wrote to P. G Tait in 1867 – the letter, whose ideas are now known as *Maxwell's Daemon*, tells of a thought experiment that seems to indicate that intelligent beings can somehow violate the second law of thermodynamics, thereby violating physics itself. Many resolutions were offered for this conundrum (for a compilation, see the book by Leff and Rex [3]), but none withstood careful scrutiny until the establishment of *Landauer's Principle* in 1961 [4] – a principle whose experimental validation happened in 2012 [5].

Maxwell's Daemon appears to violate the second law of thermodynamics by having a tiny "intelligence" observing the movement of individual particles of a gas and separating fast moving particles from slow moving ones, thereby reducing the total entropy of the system. Landauer's resolution of the daemon relied on two ideas that had taken root only a few decades earlier: the formal notion of computation (through the work of Turing [6], Church [7], and others) and the formal notion of information (through the work of Shannon [8]). Landauer reasoned that the computation done by the finite brain of the daemon involves getting information about the movement of molecules, storing that information, analyzing that information to act on it, and then — and this is the critical step — overwriting it to make room for the next computation. In other words, the computation that is manipulating information in the daemon's brain *must be thermodynamic work*, thereby bringing the daemon back into the fold of physics.

This is a strange and wonderful idea: information, physics, and computation are inextricably linked. In contrast, when the early models of computation were developed, there was no compelling reason to take the information content of computations into consideration – in fact, at that time there was no quantifiable notion of information. These models followed in the footsteps of logic where, following hundreds of years of tradition, the truth of a statement was seen as *absolute* and independent of any reasoning, understanding, or action. Statements were either true or false with no regard to any *observer* and the idea that statements had information content that should be preserved was outside the classical understanding of logic. Hence the fact that conventional logic operations such as conjunction and disjunction were logically irreversible and hence lose information was not a concern. Landauer's observation implied however that ideas in each field have consequences for the other [9, 10, 11, 12, 13, 14, 15]. To really appreciate this fact, we delve deeper

into the origin of our computational models and argue that they are essentially reflections of contemporary laws of physics.

Origins of Computational Models. Current high-level programming languages as well as current hardware are both based on the mathematical formalization of logic developed by De Morgan, Venn, Boole, and Peirce in the mid to late 1800s. Going back to Boole's 1853 book entitled *An Investigation of the Laws of Thought, on which are Founded the Mathematical Theories of Logic and Probabilities*, we find that the opening sentence of Ch. 1 is:

> The design of the following treatise is to investigate the fundamental laws of those operations of the mind by which reasoning is performed;

which clearly identifies the *source* of the logical laws as mirroring Boole's understanding of human reasoning. A few chapters later, we find:

> **Proposition IV.** That axiom of metaphysicians which is termed the principle of contradiction, and which affirms that it is impossible for any being to possess a quality, and at the same time not to possess it, is a consequence of the fundamental law of thought, whose expression is $x^2 = x$.

This "law" is reasonable in a classical world but is violated by the postulates of quantum mechanics. Although a detailed historical analysis of Boole's ideas in the light of modern physics is beyond our scope, the above quotes should convey the idea that our elementary computing notions date back to ideas that were thought reasonable in the late 1800s.

Machines that "compute" are quite old. Müller (1786) first conceived of the idea of a "difference machine," which Babbage (1819–1822) was able to construct. There are other computer precursors as well – the first stored programs were actually for looms, most notably those of Bouchon (1725) which were controlled by a paper tape, and of Jacquard (1804), controlled by chains of punched cards. But it was only in the 20th century that computer science emerged as a formal discipline. One of the pioneering works was Alan Turing's seminal paper [6] of 1936 which established the idea that computation has a formal interpretation and that all computability can be captured within a formal system. Implicit in this achievement however is the idea that abstract models of computation are just that – *abstractions of computation realized in the physical world.* Indeed, going back to Turing's 1936 article *On Computable Numbers, with an Application to the Entscheidungsproblem,* the opening sentence of Sec. 1 is:

> We have said that the computable numbers are those whose decimals are calculable by finite means [...] the justification lies in the fact that the human memory is necessarily limited.

In Sec. 9, we find:

> I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount.

It is worth noting that these assumptions are both physical (on distances) and metaphysical (on restrictions of the mind). If we take the human mind to be a physical "machine" which performs computation, then when both of the above assumptions are translated into the language of physics, they embody what is known as the "Bekenstein bound" [16], which is an upper limit on the amount of information that can be contained within a given finite region of space. A detailed historical account of these ideas in the context of modern physics is again beyond our scope. However, the quotes above, like the ones before, should convey the ideas that our theories of computation and complexity are based on some physical assumptions that Turing and others found reasonable in the 1930s.

To summarize, a major achievement of computer science has been the development of abstract models of computation that shield the discipline from rapid changes in the underlying technology. Yet, as effective as these models have been, one must note that they *embody several implicit physical assumptions* and these assumptions are based on a certain understanding of the laws of physics. Our understanding of physics

has evolved tremendously since 1900! Thus it is time to revisit these abstractions, especially with respect to quantum mechanics. Indeed one should take the physical principles underlying quantum mechanics, the most successful physical theory known to us, and adapt computation to "learn" from these principles. In the words of Girard [17]:

> In other terms, what is so good in logic that quantum physics should obey? Can't we imagine that our conceptions about logic are wrong, so wrong that they are unable to cope with the quantum miracle? [...] Instead of teaching logic to nature, it is more reasonable to learn from her. Instead of interpreting quantum into logic, we shall interpret logic into quantum.

There are, in fact, many different quantum mechanical principles which are at odds with our current models of computation. In this paper, we will focus on the previously identified principle of *reversibility*. In more detail, we will view data as an explicit representation of *information* and programs as processes that transform information in a reversible way, i.e., processes that are subject to the physical principle of *conservation of information*. We will formalize this idea and follow its consequences, which will turn out to be far reaching.

Programs as Reversible Deformations. To better understand the essence of "conservation of information" in the context of computing, we first look for analogous ideas in physics, but this time at the macro scale. Viewing information as a physical object, what does it mean to transform an object in such a way that we do not lose its fundamental character?

For rigid objects (like a chair), the only such transformations are translations and rotations. But what about something more flexible, with multiple representations, such as a water balloon? Such objects can be *deformed* in various ways, but still retain their fundamental character – as long as we do not puncture them or over-stretch them. Ignoring material characteristics (i.e. over-stretching), what is special about these deformations, as well as for translations and rotations, is that they correspond to continuous maps, with a continuous inverse. In fact, even more is true: they are analytic maps, with analytic inverses. For our purpose, the most important part is that such maps are infinitely differentiable. In other words, not only is there an inverse to the deformation, but its derivative is also invertible, and so on.

When we look around, we find many different words for related concepts: isomorphism, equivalence, sameness, equality, interchangeability, comparability, and correspondence, to name a few. Some of these are informal concepts, while others have formal mathematical meaning. More importantly, even amongst the formal concepts, there are differences – which is why there are so many of them! Because there are many such notions, we also need to walk our way through them to find the one which is "just right." Thus we seek a concept which is neither too strong nor too weak, that will express when some structured information should be treated as "the same." We can draw an analogy with topology: in topology, all point sets can always be equipped with either the discrete or the indiscrete topology, but both of these extremes are rarely useful. We will develop our working notion of "sameness" as we go through the various components that make up a programming language.

Starting from the physical perspective, whatever our notion of data is, we will be interested in programs as representing transformations of that data which are reversible. In other words, we want our programs-as-transformations to "play well" with the inherent notion of "sameness" that our data will carry. Thus we need to start by looking at what structure our data has, which will help us define an appropriate notion of a reversible program. Of course, when programs themselves are data, things do get more complicated. In the following sections, we will look at different natural classes of data, and explore the corresponding notion of reversible programs.

To summarize, we will take "the same" as a fundamental principle and derive what it means for data, programs, program transformations, as well as proofs / deductions, to be "the same" – in a manner consistent with preservation of information. This stands in stark contrast with most current approaches to reversible computation, which start from current models of computation involving irreversible operations and try to find various ways to *patch things up* so as to be reversible.

Reversible Programming Languages. The practice of programming languages is replete with *ad hoc* instances of reversible computations: database transactions, mechanisms for data provenance, checkpoints, stack and exception traces, logs, backups, rollback recoveries, version control systems, reverse engineering, software transactional memories, continuations, backtracking search, and multiple-level undo features in commercial applications. In the early nineties, Baker [18, 13] argued for a systematic, first-class, treatment of reversibility. But intensive research in full-fledged reversible models of computations and reversible programming languages was only sparked by the discovery of deep connections between physics and computation [4, 19, 1, 10, 20], and by the potential for efficient quantum computation [2].

The early developments of reversible programming languages started with a conventional programming language, e.g., an extended λ-calculus, and either

1. extended the language with a history mechanism [21, 22, 23, 24], or
2. imposed constraints on the control flow constructs to make them reversible [25].

More modern approaches recognize that reversible programming languages require a fresh approach and should be designed from first principles without the detour via conventional irreversible languages [26, 27, 28, 29].

In previous work, Carette, Bowman, James, and Sabry [30, 31, 32] introduced the Π family of typed reversible languages. As motivated above, the starting point for this development is the physical principle of *conservation of information* [33, 34] and the family of languages is designed to embrace this principle by requiring all computations to preserve information.

The fragment without recursive types is universal for reversible boolean circuits [31] and the extension with recursive types and trace operators [35] is a Turing-complete reversible language [31, 30]. While at first sight, Π too might appear *ad hoc*, it really arises naturally from an "extended" view of the Curry-Howard correspondence [32]: rather than looking at mere *inhabitation* as the main source of analogy between logic and computation, *type equivalence* becomes the source of analogy. Taking inspiration from the fact that many terms of the λ-calculus arise from Cartesian Closed Categories including, most importantly, a variety of propositional equalities and computation rules, allows us to pursue that analogy further. Some of the details of this development will be motivated and explained in the present paper.

2 Data I: Finite Sets

Most programming languages provide primitive data like booleans, characters, strings, and (bounded) numbers that are naturally modeled as finite sets. We therefore start by modeling reversible computations over finite and discrete spaces of points. Infinite sets are more subtle, and will be discussed in the conclusion.

What does it mean to deform a space of points? For example, what transformation can we do on a bag of marbles? Well, we can shuffle them around and that is the only transformation that will preserve the space. Turning to the mathematical abstraction as sets, we ask what does it mean for two finite sets to be "the same"? Well, clearly the sets $A = \{1, 2, 3\}$ and $B = \{c, d\}$ are different. Why? Well, suppose there was a transformation $f : A \to B$ that deformed A into B, and another $g : B \to A$ which undid this transformation. Since f is total, by the pigeonhole principle, two elements of A would be mapped to the same element of B. Suppose that this is 2 and 3, and that they both map to d. But $g(d)$ cannot be both 2 and 3, and so g is not the inverse of f. With just a little more work, we can show that f (and g) must be both injective and surjective. In other words, f (and g) must be a bijection between A and B. And of course this only happens when A and B have the same number of elements. More importantly, given a bijection $f : C \to D$ of finite sets C, D, there always exists another bijection $g : D \to C$ which is f's inverse. So, for finite sets, *bijections* act as reversible deformations.

This discussion is purely "semantic," in the sense that it is about the denotation of simple primitive data (sets) and their reversible deformations (bijections). We would like to reverse engineer a programming language from this denotation. But first, an obvious remark: any two sets C and D of cardinality n are always in bijective correspondence. So we can abstract away from the details of the elements of C and D

and instead choose canonical representations – in much the same way as computers choose binary words to represent everything.

Definition 2.1. *For $n \in \mathbb{N}$, denote by $[n]$ the set $\{0, 1, \ldots, n-1\}$. We will refer to $[n]$ as the canonical set with n elements.*

Bijections on $[n]$ have a specific name: permutations. As is well-known, permutations can be generated by sequential compositions of transpositions. Thus we can create a small language for writing permutations on $[n]$ as:

$$p^n \quad ::= \quad id \;\mid\; swap\ i\ j \;\mid\; p^n \fatsemi p^n$$

where $i, j : \mathbb{N}$, $i \neq j$ and $i, j < n$. Note that we could remove *id* from the language and drop the $i \neq j$ condition so that *swap* j j would represent the identity permutation.

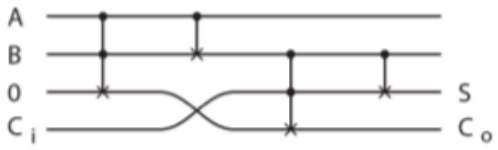
For convenience, we write $[2^n]$ for the finite set representing n-bit words with the canonical ordering for binary numbers. Thus when $n = 3$, the finite set has elements $\{0, 1, 2, 3, 4, 5, 6, 7\}$ which correspond to the 3-bit words $\{000, 001, 010, 011, 100, 101, 110, 111\}$. Although this language appears weak, it is universal for reversible boolean combinational circuits $[2^i] \to [2^i]$ with i input/output wires.

To illustrate the expressiveness of the language, we develop a few small examples. We start by writing boolean negation "not" as a permutation $[2^1] \to [2^1]$, the controlled-not gate (also known as "cnot") as a permutation $[2^2] \to [2^2]$, and the controlled-controlled-not gate (also known as "toffoli") as a permutation $[2^3] \to [2^3]$:

$$\begin{array}{rcl} \text{not} & = & swap\ 0\ 1 \\ \text{cnot} & = & swap\ 2\ 3 \\ \text{toffoli} & = & swap\ 6\ 7 \end{array}$$

The "cnot" gate operates on two bits and negates the second (the target bit) if the first one (the control bit) is 1, i.e., it swaps 10 and 11; the "toffoli" gate negates the third bit (the target bit) if both the first two bits (the control bits) are 1, i.e., it swaps 110 and 111.

There is however a subtle issue: programming in such an unstructured language is *not* compositional in the sense that using the "not" gate in a larger circuit forces us to change its implementation. Indeed if we had two bits and wanted to use "not" to negate the first bit, we would write the permutation of type $[2^2] \to [2^2]$ that permutes 00 with 10 *and* permutes 01 with 11, i.e, the permutation *swap* 0 2 ⨟ *swap* 1 3. To illustrate how inconvenient this is, consider the reversible full adder below designed by Desoete et al. [36]:



In the figure (copied from a more general paper that includes alternative designs [37]), the full adder takes 4 inputs: the two bits to add A and B, an incoming carry bit C_i, and a heap input initialized to 0 to maintain reversibility. There are four outputs: the first two are identical to the incoming bits A and B and are considered "garbage." The third bit S is the sum and the last bit C_o is the outgoing carry bit. In the notation used to describe the circuit, the $\times$ denotes boolean negation and the dots are control bits. In our reversible language, we can express this circuit as the following permutation of type $[2^4] \to [2^4]$:

$$\begin{array}{ll} swap\ 12\ 14 \fatsemi swap\ 13\ 15 \fatsemi & \text{toffoli} \\ swap\ 8\ 12 \fatsemi swap\ 9\ 14 \fatsemi swap\ 10\ 13 \fatsemi swap\ 11\ 15 \fatsemi & \text{cnot and swap} \\ swap\ 6\ 7 \fatsemi swap\ 14\ 15 \fatsemi & \text{toffoli} \\ swap\ 4\ 6 \fatsemi swap\ 5\ 7 \fatsemi swap\ 12\ 14 \fatsemi swap\ 13\ 15 & \text{cnot} \end{array} \tag{1}$$

Note how the implementation of *cnot* as a permutation $[2^2] \to [2^2]$ cannot be directly reused in the larger circuit $[2^4] \to [2^4]$.

For such reasons, in programming practice we are interested in structured data and compositional abstractions, which will be the subject of the next section. What we do learn from this short investigation using untyped and unstructured sets is what the "*purely operational*" view of the theory would be. In particular, it tells us that permutations are an inescapable part of the fabric of reversible computing. However as permutations are untyped, and act on the canonicalized version of n-element sets (i.e. those sets where all the structure has been forgotten), these are a rather pale shadow of the rich tapestry of information-preserving transformations of structured data, which we investigate next.

3 Data II: Structured Finite Types

Instead of spaces (aka discrete sets) consisting solely of unstructured isolated points, we now investigate structured spaces built from sums and products of elementary spaces. This structure corresponds to the building blocks of type theory which are: the empty type ($\bot$), the unit type ($\top$), the sum type ($\uplus$), and the product ($*$) type. Before getting into the formal theory, let's consider possible deformations on the space $(\top \uplus \bot) * (\top \uplus \top)$. This space is the product of two subspaces: the subspace $(\top \uplus \bot)$ which itself is the sum of the space $\top$ containing one element `tt` and the empty space $\bot$ and the subspace $(\top \uplus \top)$ which is the sum of two spaces each containing the one element `tt`. First, as discussed in the previous section, any deformation of this space must at least preserve the number of elements: we can neither create nor destroy points during any continuous deformation. Seeing that the number of elements in our example space is 2, a reasonable hypothesis is that we can deform the space above to any other space with 2 elements such as $\top \uplus \top$ or $\top \uplus (\top \uplus \bot)$. What this really means is that we are treating the sum and product structure as malleable. For example, imagining a product structure as arranged in a grid; by "stretching" we can turn it in to a sum structure arranged in a line. We can also change the orientation of the grid by exchanging the axes, as well as do other transformations — as long as we preserve the number of points. Of course, it is not a priori clear that this necessary requirement is also sufficient. Making this intuition precise will be the topic of this section.

3.1 A Model of Type Equivalences

We now want a proper mathematical description of this idea. Our goal is a denotational semantics on types which makes types that have the same number of points be equivalent types. First we note that the structure of types has a nice correspondence (Curry-Howard) to logic:

Logic	Types
false	$\bot$
true	$\top$
$\wedge$	$*$
$\vee$	$\uplus$

This correspondence is rather fruitful. As logical expressions form a commutative semiring, we would expect that types too form a commutative semiring. And indeed they do – at least up to *type isomorphism*. The natural numbers $\mathbb{N}$ are another commutative semiring; it will turn out that, even though the Curry-Howard correspondence has been extremely fruitful for programming language research, it is $\mathbb{N}$ which will be a better model for finite structured types as the corresponding commutative semiring captures the familiar numerical identities that preserve the number of points in the types.

Definition 3.1. *A* commutative semiring *(sometimes called a* commutative rig *— commutative ring without negative elements)* $(R, 0, 1, +, \cdot)$ *consists of a set* R*, two distinguished elements of* R *named* 0 *and* 1*, and two*

binary operations $+$ and $\cdot$, satisfying the following relations for any $a, b, c \in R$:

$$\begin{aligned} 0 + a &= a \\ a + b &= b + a \\ a + (b + c) &= (a + b) + c \end{aligned}$$

$$\begin{aligned} 1 \cdot a &= a \\ a \cdot b &= b \cdot a \\ a \cdot (b \cdot c) &= (a \cdot b) \cdot c \end{aligned}$$

$$\begin{aligned} 0 \cdot a &= 0 \\ (a + b) \cdot c &= (a \cdot c) + (b \cdot c) \end{aligned}$$

Proposition 3.1. *The structure $(\{false, true\}, false, true, \vee, \wedge)$ is a commutative semiring.*

We would like to adapt the commutative semiring definition to the setting of structured types. First, types do not naturally want to be put together into a "set." This can be fixed if we replace the set R with a universe U, and replace the set membership $0 \in R$ with the typing judgement $\bot : U$ (and similarly for the other items). Our next instinct would be to similarly replace $=$ with a type $A \equiv B$ that asserts that A and B are *propositionally equal*, i.e. reduce to equivalent type-denoting expressions under the rules of the host type system. This is however not true: the proposition $A * B \equiv B * A$ is not normally[1] provable for arbitrary types A and B. But it should be clear that $A * B$ and $B * A$ contain equivalent information. In other words, we would like to be able to witness that $A * B$ can be reversibly deformed into $B * A$, and vice-versa, which motivates the introduction of type *equivalences*. To do this, we need a few important auxiliary concepts.

Definition 3.2 (Propositional Equivalence)**.** *Two expressions a, b of type A are* propositionally equal *if their normal forms are equivalent under the rules of the host type system.*

In Martin-Löf Type Theory, normal forms mean $\beta\eta$-long normal forms under α-equivalence. In other words, expressions are evaluated as much as possible (β-reduced), all functions are fully applied (η-long), and the exact names of bound variables are irrelevant (α-equivalence). Note that the above definition applies equally well to expressions that denote values and expressions that denote types.

Definition 3.3 (Homotopy)**.** *Two functions $f, g : A \to B$ are* homotopic *if $\forall x : A. f(x) \equiv g(x)$. We denote this $f \sim g$.*

It is easy to prove that homotopies (for any given function space $A \to B$) are an equivalence relation. The simplest definition of the data which makes up an equivalence is the following.

Definition 3.4 (Quasi-inverse)**.** *For a function $f : A \to B$, a* quasi-inverse *is a triple (g, α, β), consisting of a function $g : B \to A$ and two homotopies $\alpha : f \circ g \sim \mathrm{id}_B$ and $\beta : g \circ f \sim \mathrm{id}_A$.*

Definition 3.5 (Equivalence of types)**.** *Two types A and B are equivalent $A \simeq B$ if there exists a function $f : A \to B$ together with a quasi-inverse for f.*

Why *quasi*? The reasons are beyond our scope, but the interested reader can read Sec. 2.4 and Ch. 4 in the Homotopy Type Theory (HoTT) book [38]. There are several conceptually different, but equivalent, "better" definitions. We record just one here:

Definition 3.6 (Bi-invertibility)**.** *For a function $f : A \to B$, a* bi-inverse *is a pair of functions $g, h : B \to A$ and two homotopies $\alpha : f \circ g \sim \mathrm{id}_B$ and $\beta : h \circ f \sim \mathrm{id}_A$.*

We can then replace quasi-inverse with bi-invertibility in the definition of type equivalence. The differences will not matter to us here.

[1] Except in univalent type theory where equivalent types are identified.

$$
\begin{array}{rcl}
A & \simeq & A \\
\\
\bot \uplus A & \simeq & A \\
A \uplus B & \simeq & B \uplus A \\
A \uplus (B \uplus C) & \simeq & (A \uplus B) \uplus C \\
\\
\top * A & \simeq & A \\
A * B & \simeq & B * A \\
A * (B * C) & \simeq & (A * B) * C \\
\\
\bot * A & \simeq & \bot \\
(A \uplus B) * C & \simeq & (A * C) \uplus (B * C)
\end{array}
$$

Figure 1: Type isomorphisms.

We are now in position to describe the commutative semiring structure for types. After replacing the set R with a universe U, we also replace the algebraic use of $=$ in Def. 3.1 by the type equivalence relation $\simeq$. With this change, we can indeed prove that types (with $\bot, \top, \uplus, *$) form a commutative semiring. The reader familiar with universal algebra should pause and ponder a bit about what we have done. We have lifted *equality* from being in the signature of the ambient logic and instead put it in the signature of the algebraic structure of interest. In simpler terms, we shift equality from having a privileged status in our meta-theory, to being just another symbol (denoting an equivalence relation) in our theory. The understanding that equality is not an absolute concept has recently been an area of active research in mechanized mathematics — although the concepts of intensional versus extensional equality go back to Frege and Russell.

If we revisit the Curry-Howard correspondence, we notice one more issue. In logic, it is true that $A \vee A = A$ and $A \wedge A = A$. However, neither $A \uplus A$ nor $A * A$ are equivalent to A. They are however *equi-inhabited*. This is a fancy way of saying

$$A \uplus A \text{ is inhabited} \qquad \Leftrightarrow \qquad A \text{ is inhabited}$$

The above is the real *essence* of the Curry-Howard correspondence. In other words, classical Curry-Howard tells us about *logical equivalence* of types. This is even a constructive statement: there are indeed functions $f : A \uplus A \to A$ and $g : A \to A \uplus A$; however, they are not inverses.

So mere inhabitation falls far short of our goals of being able to smoothly deform from one type to another. Let us thus analyze the crux of the "problem." In logic, we have that $\wedge$ and $\vee$ are both *idempotent*: this is the property of any binary operation $\circ$ where $\forall a. a \circ a = a$. And it should be clear that an idempotent operations is a *forgetful* operation: its input has two copies of a, but its output, only one. On the type side, something more subtle happens. Consider $\top \uplus \top$ versus $\top$; the first has exactly *two* proofs of inhabitation ($\mathsf{left}\ \mathtt{tt}$ and $\mathsf{right}\ \mathtt{tt}$) while the second only one ($\mathtt{tt}$). These cannot be put in bijective correspondence. Even though the "payload" $\mathtt{tt}$ is the same, forgetting $\mathtt{left}$ (or $\mathtt{right}$) throws away information – something we have expressly disallowed. Yes, this should remind you of Maxwell's daemon: even though the data is the same, they are tagged differently, and these tags are indeed information, and their information content must be preserved.

Nevertheless, the Curry-Howard correspondence still has some force. We know that the inhabitants of types formed with with $\bot, \top, \uplus, *$ form a commutative semiring. What we want to know is, which types are equivalent? From a commutative semiring perspective, this amounts to asking what terms are equal. We have a set of generators for those equations, namely those in Def. 3.1. What we thus need is to create 8 pairs of mutually inverse functions which witness these identities. For concreteness, we show the signatures in Fig. 1.

From category theory, we are informed of the following privilege enjoyed by the natural numbers $\mathbb{N}$:

Theorem 3.1. *The semiring* $(\mathbb{N}, 0, 1, +, \cdot)$ *is* initial *in the category of semirings and semiring homomor-*

$$
\begin{array}{rrcll}
id\leftrightarrow: & t & \leftrightarrow & t & : id\leftrightarrow \\
\\
unite_+ l: & 0+t & \leftrightarrow & t & : uniti_+ l \\
swap_+: & t_1+t_2 & \leftrightarrow & t_2+t_1 & : swap_+ \\
assocl_+: & t_1+(t_2+t_3) & \leftrightarrow & (t_1+t_2)+t_3 & : assocr_+ \\
\\
unite_\times l: & 1\times t & \leftrightarrow & t & : uniti_\times l \\
swap_\times: & t_1\times t_2 & \leftrightarrow & t_2\times t_1 & : swap_\times \\
assocl_\times: & t_1\times(t_2\times t_3) & \leftrightarrow & (t_1\times t_2)\times t_3 & : assocr_\times \\
\\
absorbr: & 0\times t & \leftrightarrow & 0 & : factorzl \\
dist: & (t_1+t_2)\times t_3 & \leftrightarrow & (t_1\times t_3)+(t_2\times t_3) & : factor
\end{array}
$$

Figure 2: Π-terms.

$$
\frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_2 \leftrightarrow t_3}{\vdash c_1 \odot c_2 : t_1 \leftrightarrow t_3}
\qquad
\frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \oplus c_2 : t_1 + t_3 \leftrightarrow t_2 + t_4}
$$

$$
\frac{\vdash c_1 : t_1 \leftrightarrow t_2 \quad \vdash c_2 : t_3 \leftrightarrow t_4}{\vdash c_1 \otimes c_2 : t_1 \times t_3 \leftrightarrow t_2 \times t_4}
$$

Figure 3: Π-combinators.

phisms.

In other words, for any semiring S, there is a homomorphism from $\mathbb{N}$ into S. But $\mathbb{N}$ is also the "counting" semiring, which formalizes the notion of cardinality of finite discrete sets.

The previous section on finite sets, along with the reasoning above, thus leads us to posit that the correct denotational semantics for finite discrete types is that of the semiring $(\mathbb{N}, 0, 1, +, \cdot)$. It is worth noting that equality in this semiring is intensional (i.e. two things are equal if and only if they are identical after evaluation), unlike that for types.

3.2 A Language of Type Equivalences

We now have in our hands our desired denotational semantics for types. We want to create a programming language, which we call Π, such that the types and type combinators map to $\bot, \top, \uplus, *$, and such that we have ground terms whose denotation are all 16 type isomorphisms of Fig. 1. This is rather straightforward, as we can simply do this literally. To make the analogy with commutative semirings stand out even more, we will use $0, 1, +$, and $\times$ at the type level, and will denote "equivalence" by $\leftrightarrow$. Thus Fig. 2 shows the "constants" of the language. As these all come in symmetric pairs (some of which are self-symmetric), we give names for both directions. Note how we have continued with the spirit of Curry-Howard: the terms of Π are *proof terms*, but rather than being witnesses of inhabitation, they are witnesses of equivalences. Thus we get an unexpected programming language design:

The proof terms denoting commutative semiring equivalences induce the terms of Π.

Of course, one does not get a programming language with just typed constants! There is a need to perform multiple equivalences. There are in fact three ways to do this: sequential composition $\odot$, choice composition $\oplus$ (sometimes called juxtaposition), and parallel composition $\otimes$. See Fig. 3 for the signatures. The construction $c_1 \odot c_2$ corresponds to performing c_1 first, then c_2, and is the usual notion of composition – and corresponds to $\fatsemi$ of the language of permutations of Sec. 2. The construction $c_1 \oplus c_2$ chooses to perform c_1 or c_2 depending

$$\frac{\vdash c_1 : t_1 \leftrightarrow t_2}{\vdash\ !\ c_1 : t_2 \leftrightarrow t_1}$$

Figure 4: Derived Π-combinator.

$$\begin{array}{rrcll}
\mathit{unite}_+ r : & t + 0 & \leftrightarrow & t & : \mathit{uniti}_+ r \\
\mathit{unite}_\times r : & t \times 1 & \leftrightarrow & t & : \mathit{uniti}_\times r \\
\\
\mathit{absorbl} : & t \times 0 & \leftrightarrow & 0 & : \mathit{factorzr} \\
\mathit{distl} : & t_1 \times (t_2 + t_3) & \leftrightarrow & (t_1 \times t_2) + (t_1 \times t_3) & : \mathit{factorl}
\end{array}$$

Figure 5: Additional Π-terms.

on whether the input is labelled left or right respectively. Finally the construction $c_1 \otimes c_2$ operates on a product structure, and applies c_1 to the first component and c_2 to the second. The language of permutations lacked the ability to combine permutations by taking sums and products, which led to the awkward non-compositional programming style illustrated in the full adder example (Eq. 1).

Thus the denotation of the Π terms *should* be permutations. But given types A and B denoting $[m]$ and $[n]$ respectively, what are $A \uplus B$ and $A * B$? They correspond exactly to $[m + n]$ and $[m * n]$! Geometrically, this corresponds to concatenation for $A + B$, i.e. lining up the elements of A first, and then those of B. For $A * B$, one can picture this as lining up the elements of A horizontally, those of B vertically and perpendicular to those of A, and filling in the square with pairs of elements from A and B; if one re-numbers these sequentially, reading row-wise, this gives an enumeration of $[m * n]$.

From here, it is easy to see what, for example, $c_1 \oplus c_2$ must be, operationally: from a permutation on $[m]$ and another on $[n]$, create a permutation on $[m + n]$ by having c_1 operate on the first m elements of $A + B$, and c_2 operate on the last n elements. Similarly, swap_+ switches the roles of A and B, and thus corresponds to $[n + m]$. Note how we "recover" the commutativity of natural number addition from this type isomorphism. Geometrically, $\mathit{swap}_\times$ is also rather interesting: it corresponds to matrix transpose! Furthermore, in this representations, some combinators like $\mathit{unite}_+ l$ and assocl_+ are identity operations: the underlying representations are not merely isomorphic, they are definitionally equal. In other words, the passage to $\mathbb{N}$ erases some structural information.

Embedded in our definition of Π is a conscious design decision: to make the terms of Π *syntactically* reversible. In other words, to every Π constant, there is another Π constant which is its inverse. As this is used frequently, we give it the short name !, and its type is given in Fig. 4. This combinator is *defined*, by pattern matching on the syntax of its argument and structural recursion.

This is not the only choice. Another would be to add a *flip* combinator to the language; we could then remove quite a few combinators as redundant. The drawback is that many programs in Π become longer. Furthermore, some of the symmetry at "higher levels" (see next section) is also lost. Since the extra burden of language definition and of proofs is quite low, we prefer the structural symmetry over a minimalistic language definition.

We also make a second design decision, which is to make the Π language itself symmetric in another sense: we want both left and right introduction/elimination rules for units, 0 absorption and distributivity. Specifically, we add the Π-terms of Fig. 5 to our language. These are redundant because of swap_+ and $\mathit{swap}_\times$, but will later enable shorter programs and more elegant presentation of program transformations.

This set of isomorphisms is known to be sound and complete [39, 40] for isomorphisms of finite types. Furthermore, it is also universal for hardware combinational circuits [31].

3.3 Operational Semantics

To give an operational semantics to Π, we are mainly missing a notation for *values*.

Definition 3.1. *(Syntax of values of* Π*)*

$$values, v \quad ::= \quad () \mid left\ v \mid right\ v \mid (v, v)$$

Given a program $c : b_1 \leftrightarrow b_2$ in Π, we can run it by supplying it with a value $v_1 : b_1$. The evaluation rules $c\ v_1 \mapsto v_2$ are given below.

Definition 3.2. *(Operational Semantics for* Π*)*

Identity:

$$id\leftrightarrow \quad v \quad \mapsto \quad v$$

Additive fragment:

$$\begin{array}{rlcl}
unite_+l & (right\ v) & \mapsto & v \\
uniti_+l & v & \mapsto & right\ v \\
unite_+r & (left\ v) & \mapsto & v \\
uniti_+r & v & \mapsto & left\ v \\
swap_+ & (left\ v) & \mapsto & right\ v \\
swap_+ & (right\ v) & \mapsto & left\ v \\
assocl_+ & (left\ v_1) & \mapsto & left\ (left\ v_1) \\
assocl_+ & (right\ (left\ v_2)) & \mapsto & left\ (right\ v_2) \\
assocl_+ & (right\ (right\ v_3)) & \mapsto & right\ v_3 \\
assocr_+ & (left\ (left\ v_1)) & \mapsto & left\ v_1 \\
assocr_+ & (left\ (right\ v_2)) & \mapsto & right\ (left\ v_2) \\
assocr_+ & (right\ v_3) & \mapsto & right\ (right\ v_3)
\end{array}$$

Multiplicative fragment:

$$\begin{array}{rlcl}
unite_\times l & ((), v) & \mapsto & v \\
uniti_\times l & v & \mapsto & ((), v) \\
unite_\times r & (v, ()) & \mapsto & v \\
uniti_\times r & v & \mapsto & (v, ()) \\
swap_\times & (v_1, v_2) & \mapsto & (v_2, v_1) \\
assocl_\times & (v_1, (v_2, v_3)) & \mapsto & ((v_1, v_2), v_3) \\
assocr_\times & ((v_1, v_2), v_3) & \mapsto & (v_1, (v_2, v_3)) \\
absorbr & (v_1, v_2) & \mapsto & v_1
\end{array}$$

Distributivity and factoring:

$$\begin{array}{rlcl}
dist & (left\ v_1, v_3) & \mapsto & left\ (v_1, v_3) \\
dist & (right\ v_2, v_3) & \mapsto & right\ (v_2, v_3) \\
distl & (v_1, left\ v_2) & \mapsto & left\ (v_1, v_2) \\
distl & (v_1, right\ v_3) & \mapsto & right\ (v_1, v_3) \\
factor & (left\ (v_1, v_3)) & \mapsto & (left\ v_1, v_3) \\
factor & (right\ (v_2, v_3)) & \mapsto & (right\ v_2, v_3) \\
factorl & (left\ (v_1, v_2)) & \mapsto & (v_1, left\ v_2) \\
factorl & (right\ (v_1, v_3)) & \mapsto & (v_1, right\ v_3) \\
absorbl & (v_1, v_2) & \mapsto & v_2
\end{array}$$

The evaluation rules of the composition combinators are given below:

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

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

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

Since there are no values that have the type 0, the reductions for the combinators $unite_+l$, $uniti_+l$, $unite_+r$, and $uniti_+r$ omit the impossible cases. *factorzr* and *factorzl* likewise do not appear as they have no possible cases at all. However, *absorbr* and *absorbl* are treated slightly differently: rather than *eagerly* assuming they are impossible, the purported inhabitant of 0 given on one side is passed on to the other side. The reason for this choice will have to wait for Sec. 4.2 when we explain some higher-level symmetries (see Fig. 13).

As we mentioned before, ! is a defined combinator.

Definition 3.3 (Adjoint, ! c)**.** *The adjoint of a combinator c is defined as follows:*

- *For primitive isomorphisms c, ! c is given by its inverse from Figs. 2 and 5.*
- $!(c_1 \otimes c_2) = !c_1 \otimes\ !c_2$
- $!(c_1 \oplus c_2) = !c_1 \oplus\ !c_2$
- $!(c_1 \odot c_2) = !c_2 \odot\ !c_1$. *(Note that the order of combinators has been reversed).*

We can further define that two combinators are *observationally equivalent* if on all values of their common domain, they evaluate to identical values. More precisely, we will say that for combinators $c_1, c_2 : b_1 \leftrightarrow b_2$, $c_1 = c_2$ whenever:

$$\forall\ v_1 : b_1, v_2 : b_2.\ \ c_1\ v_1 \mapsto v_2 \text{ if and only if } c_2\ v_1 \mapsto v_2$$

Each type b has a size $|b|$ defined in the obvious way. We had previously established that for any natural number n, there is a canonical set of size n, which we denoted $[n]$. Furthermore, we can also define a canonical *type* of that size, which we will denote $\sharp\, b$, i.e. $\sharp\, b$ is a canonical type of size $|b|$.

Definition 3.4. *($\sharp$). By recursion on $|b|$. First define τ that maps numeric sizes to their corresponding types. We will revert to using type notation for greater clarity of this definition:*

$$\begin{array}{rcl} \tau\ (0) & = & \bot \\ \tau\ (1+n) & = & \top \uplus \tau\ (n) \end{array}$$

so that we can define $\sharp\, b = \tau\ |b|$.

We are now ready to go further and establish that there is always an equivalence between a type and the canonical type of the same size.

Proposition 3.5. *For any type b there exists an isomorphism $b \leftrightarrow \sharp\, b$.*

Proof. The fact that such an isomorphism exists is evident from the definition of size and what it means for two types to be isomorphic. While many equivalent constructions are possible for any type b, one such construction is given by $[\![b]\!]$:

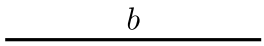
$$\begin{array}{rcl} [\![0]\!] & = & id\leftrightarrow \\ [\![1]\!] & = & id\leftrightarrow \\ [\![1{+}b]\!] & = & id\leftrightarrow \oplus\ [\![b]\!] \\ [\![(b_1{+}b_2){+}b_3]\!] & = & assocr_+ \odot [\![b_1{+}(b_2{+}b_3)]\!] \\ [\![b_1{+}b_2]\!] & = & ([\![b_1]\!] \oplus id\leftrightarrow) \odot [\![\sharp\, b_1{+}b_2]\!] \\ [\![0{\times}b_2]\!] & = & absorbr \\ [\![1{\times}b_2]\!] & = & unite_\times l \odot [\![b_2]\!] \\ [\![(b_1{\times}b_2){\times}b_3]\!] & = & assocr_\times \odot [\![b_1{\times}(b_2{\times}b_3)]\!] \\ [\![(b_1{+}b_2){\times}b_3]\!] & = & dist \odot [\![b_1{\times}b_3{+}b_2{\times}b_3]\!] \end{array}$$

□

3.4 Graphical Language

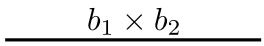
Combinators of Π can be written in terms of the operators described previously or via a graphical language similar in spirit to those developed for Geometry of Interaction [41] and string diagrams for category theory [42, 43]. Modulo some conventions and shorthand we describe here, the wiring diagrams are equivalent to the operator based (syntactic) description of programs. Π combinators expressed in this graphical language look like "wiring diagrams." Values take the form of "particles" that flow along the wires. Computation is expressed by the flow of particles.

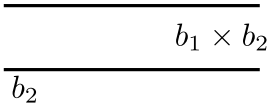
- The simplest sort of diagram is the $id\leftrightarrow\colon b \leftrightarrow b$ combinator which is simply represented as a wire labeled by its type b. In more complex diagrams, if the type of a wire is obvious from the context, it may be omitted.



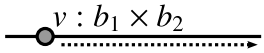
  Values flow from left to right in the graphical language of Π. When tracing a computation, one might imagine a value v of type b on the wire, as shown below.

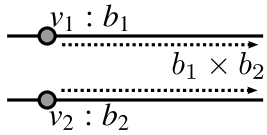
- The product type $b_1 \times b_2$ may be represented both as one wire labeled $b_1 \times b_2$ or by two parallel wires labeled b_1 and b_2. Both representations may be used interchangeably.



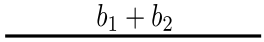


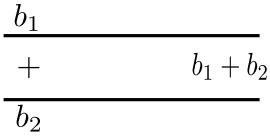
  When tracing execution using particles, one should think of one particle on each wire or alternatively as in folklore in the literature on monoidal categories as a "wave."



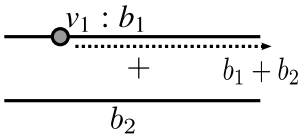


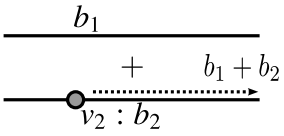
- Sum types may similarly be represented using using parallel wires with a + operator between them.



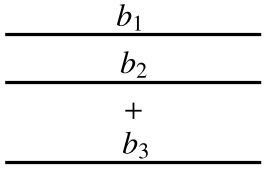


  When tracing the execution of b_1+b_2 represented by one wire, one can think of a value of the form *left* v_1 or *right* v_2 as flowing on the wire, where $v_1 : b_1$ and $v_2 : b_2$. When tracing the execution of two additive wires, a value can reside on only one of the two wires.

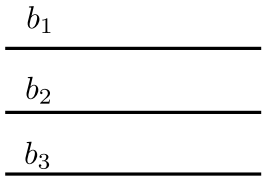




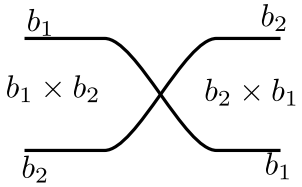
- When representing complex types like $(b_1 \times b_2) + b_3$ some visual grouping of the wires may be done to aid readability. The exact type however will always be clarified by the context of the diagram.

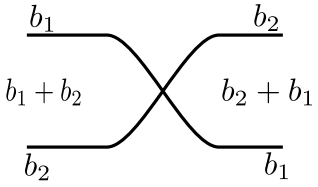


- Associativity is entirely skipped in the graphical language. Hence three parallel wires may be inferred as $b_1 \times (b_2 \times b_3)$ or $(b_1 \times b_2) \times b_3$, based on the context. This is much like handling of associativity in the graphical representations of categories as well as that for monoidal categories.

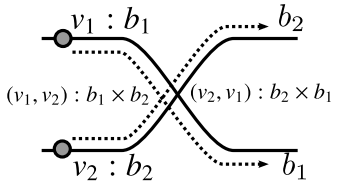


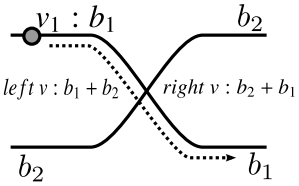
- Commutativity is represented by crisscrossing wires.





When tracing the execution of $b_1 + b_2$ represented by one wire, one can think of a value of the form *left* v_1 or *right* v_2 as flowing on the wire, where $v_1 : b_1$ and $v_2 : b_2$. By visually tracking the flow of particles on the wires, one can verify that the expected types for commutativity are satisfied.





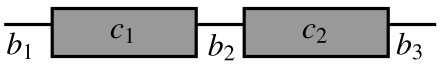
- The morphisms that witness that 0 and 1 are the additive and multiplicative units are represented as shown below. Note that since there is no value of type 0, there can be no particle on a wire of type 0. Also since the monoidal units can be freely introduced and eliminated, sometimes they are omitted. However, as this is in fact dangerous, as explained by [42], we will err on the side of including them.

- Distributivity and factoring are represented using the dual boxes shown below:

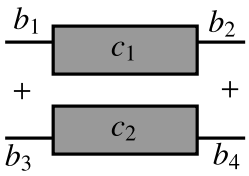
Distributivity and factoring are interesting because they represent interactions between sum and pair types. Distributivity should essentially be thought of as a multiplexer that redirects the flow of $v : b$ depending on what value inhabits the type b_1+b_2, as shown below.

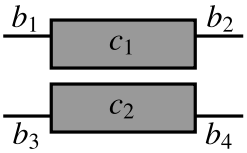
Factoring is the corresponding adjoint operation.

- Combinators can be composed in series ($c_1 \odot c_2$) or parallel. Sequential (series) composition corresponds to connecting the output of one combinator to the input of the next.



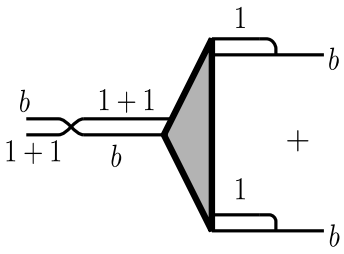
There are two forms of parallel composition – combinators can be combined additively $c_1 \oplus c_2$ (shown on the left) or multiplicatively $c_1 \otimes c_2$ (shown on the right).





Example. As an example consider the wiring diagram of the combinator c below:

$$\begin{array}{lcl} c & : & b{\times}(1{+}1) \leftrightarrow b{+}b \\ c & = & swap_{\times} \odot dist \odot (unite_{\times} l \oplus unite_{\times} l) \end{array}$$



3.5 Denotational Semantics

Fig. 1 introduces our desired denotational semantics, and Sec. 3.3 is a direct definition of an operational semantics. One obvious question arises: do these correspond?

We can certainly associate to each Π combinator an equivalence between the denotation of each type[2]:

$$\mathsf{c2equiv} : \{t_1\ t_2 : \mathsf{U}\} \to (c : t_1 \leftrightarrow t_2) \to [\![\ t_1\ ]\!] \simeq [\![\ t_2\ ]\!]$$

And as such an equivalence contains a function as its first component, we can compare if our operational semantics and denotational semantics match. And they do:

$$\mathsf{lemma0} : \{t_1\ t_2 : \mathsf{U}\} \to (c : t_1 \leftrightarrow t_2) \to (v : [\![\ t_1\ ]\!]) \to \mathsf{eval}\ c\ v \equiv \mathsf{proj}_1\ (\mathsf{c2equiv}\ c)\ v$$

We can similarly hand-write a backwards evaluator, prove that it is indeed a proper backwards evaluator, and finally show that it agrees with the reverse equivalence.

3.6 Examples

At first, it is not immediately clear that a programming language in which information is preserved could model choice. We recall a quote by Minsky communicating this concern:

> Ed Fredkin pursued the idea that information must be finite in density. One day, he announced that things must be even more simple than that. He said that he was going to assume that information itself is conserved. "You're out of you mind, Ed." I pronounced. "That's completely ridiculous. Nothing could happen in such a world. There couldn't even be logical gates. No decisions could ever be made." But when Fredkin gets one of his ideas, he's quite immune to objections like that; indeed, they fuel him with energy. Soon he went on to assume that information processing must also be reversible — and invented what's now called the Fredkin gate [33].

We will however show that one can program all logical gates in Π. We will start with a few simple examples and then discuss the expressiveness of the language and its properties.

[2]This is extracted from the Agda formalization of this work, which has been reported on in a previous paper [32].

Booleans Let us start with encoding booleans. We use the type 1+1 to represent booleans with *left* () representing *true* and *right* () representing *false*. Boolean negation is straightforward to define:

$not : bool \leftrightarrow bool$

$not = swap_+$

It is easy to verify that *not* changes *true* to *false* and vice versa.

Bit Vectors. We can represent n-bit words using an n-ary product of *bool*s. For example, we can represent a 3-bit word, $word_3$, using the type $bool \times (bool \times bool)$. We can perform various operations on these 3-bit words using combinators in Π. For instance the bitwise *not* operation is the parallel composition of three *not* operations:

$not_{word_3} :: word_3 \leftrightarrow word_3$

$not_{word_3} = not \times (not \times not)$

We can express a 3-bit word reversal operation as follows:

$reverse : word_3 \leftrightarrow word_3$

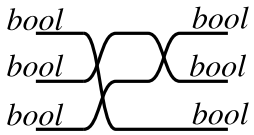
$reverse = swap_\times \odot (swap_\times \otimes id\leftrightarrow) \odot assocr_\times$

We can check that *reverse* does the right thing by applying it to a value $(v_1, (v_2, v_3))$ and writing out the full derivation tree of the reduction. The combinator *reverse*, like many others we will see in this paper, is formed by sequentially composing several simpler combinators. Instead of presenting the operation of *reverse* as a derivation tree, it is easier (purely for presentation reasons) to flatten the tree into a sequence of reductions as caused by each component. Such a sequence of reductions is given below:

$$\begin{array}{rl} & (v_1, (v_2, v_3)) \\ swap_\times & ((v_2, v_3), v_1) \\ swap_\times \otimes id\leftrightarrow & ((v_3, v_2), v_1) \\ assocr_\times & (v_3, (v_2, v_1)) \end{array}$$

On the first line is the initial value. On each subsequent line is a fragment of the *reverse* combinator and the value that results from applying this combinator to the value on the previous line. For example, $swap_\times$ transforms $(v_1, (v_2, v_3))$ to $((v_2, v_3), v_1)$. On the last line we see the expected result with the bits in reverse order.

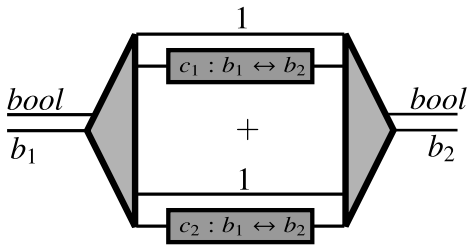
We can also draw out the graphical representation of the 3-bit reverse combinator. In the graphical representation, it is clear that the combinator achieves the required shuffling.



Conditionals. Even though Π lacks conditional expressions, they are expressible using the distributivity and factoring laws. The diagrammatic representation of *dist* shows that it redirects the flow of a value $v : b$ based on the value of another one of type $b_1 + b_2$. If we choose 1+1 to be *bool* and apply either $c_1 : b_1 \leftrightarrow b_2$ or $c_2 : b_1 \leftrightarrow b_2$ to the value v, then we essentially have an 'if' expression.

$if_{c_1,c_2} : bool \times b_1 \leftrightarrow bool \times b_2$

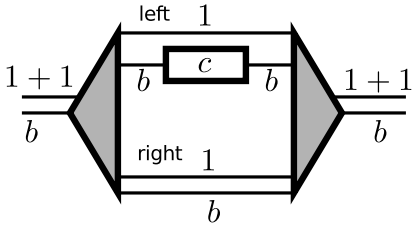
$if_{c_1,c_2} = dist \odot ((id\leftrightarrow \otimes c_1) + (id\leftrightarrow \otimes c_2)) \odot factor$



The diagram above shows the input value of type $(1+1)\times b_1$ processed by the distribute operator *dist*, which converts it into a value of type $(1\times b_1)+(1\times b_1)$. In the *left* branch, which corresponds to the case when the boolean is *true* (i.e. the value was *left* ()), the combinator c_1 is applied to the value of type b_1. The right branch which corresponds to the boolean being *false* passes the value of type b_1 through the combinator c_2. The inverse of *dist*, namely *factor* is applied to get the final result of type $(1+1)\times b_2$.

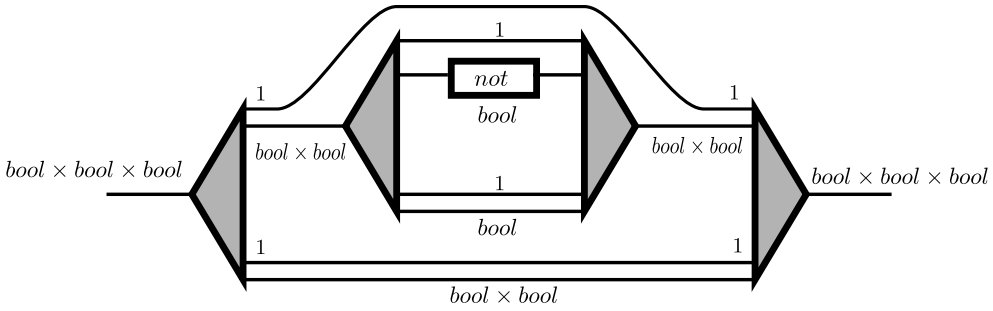
Logic Gates There are several universal primitives for conventional (irreversible) hardware circuits, such as *nand* and *fanout*. In the case of reversible hardware circuits, the canonical universal primitive is the Toffoli gate [1]. The Toffoli gate takes three boolean inputs: if the first two inputs are *true* then the third bit is negated. In a traditional language, the Toffoli gate would be most conveniently expressed as a conditional expression like:
$toffoli(v_1, v_2, v_3) = if\ (v_1\ and\ v_2)\ then\ (v_1, v_2, not(v_3))\ else\ (v_1, v_2, v_3)$

We will derive Toffoli gate in Π by first deriving a simpler logic gate called *cnot*. Consider a one-armed version, if_c, of the conditional derived above. If the *bool* is *true*, the value of type b is modified by the operator c.



By choosing b to be *bool* and c to be *not*, we have the combinator $if_{not} : bool\times bool \leftrightarrow bool\times bool$ which negates its second argument if the first argument is *true*. This gate if_{not} is often referred to as the *cnot* gate[1].

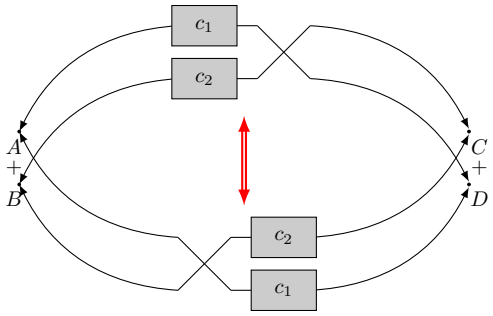
If we iterate this construction once more, the resulting combinator if_{cnot} has type $bool\times(bool\times bool) \leftrightarrow bool\times(bool\times bool)$. The resulting gate checks the first argument and if it is *true*, proceeds to check the second argument. If that is also *true* then it will negate the third argument. Thus if_{cnot} is the required Toffoli gate.



4 Data III: Reversible Programs between Reversible Programs

In the previous sections, we examined equivalences between conventional data structures, i.e., sets of values and structured trees of values. We now consider a richer but foundational notion of data: programs themselves. Indeed, universal computation models crucially rely on the fact that *programs are (or can be encoded as) data*, e.g., a Turing machine can be encoded as a string that another Turing machine (or even the same machine) can manipulate. Similarly, first-class functions are the *only* values in the λ-calculus. In our setting, the programs developed in the previous section are reversible deformations between structured finite types. We now ask whether these programs can themselves be subject to (higher-level) reversible deformations?

Before developing the theory, let's consider a small example consisting of two deformations between the types $A + B$ and $C + D$:



The top path is the Π program $(c_1 \oplus c_2) \odot \mathit{swap}_+$ which deforms the type A by c_1, deforms the type B by c_2, and deforms the resulting space by a twist that exchanges the two injections into the sum type. The bottom path performs the twist first and then deforms the type A by c_1 and the type B by c_2 as before. One could imagine the paths are physical *elastic* wires in 3 space, where the deformations c_1 and c_2 as arbitrary deformations on these wires, and the twists do not touch but are in fact well-separated. Then, holding the points A, B, C, and D fixed, it is possible to imagine sliding c_1 and c_2 from the top wire rightward past the twist, and then using the elasticity of the wires, pull the twist back to line up with that of the bottom — thus making both parts of the diagram identical. Each of these moves can be undone (reversed), and doing so would take the bottom part of the diagram into the top part. In other words, there exists a deformation of the program $(c_1 \oplus c_2) \odot \mathit{swap}_+$ to the program $\mathit{swap}_+ \odot (c_2 \oplus c_1)$. We can also show that this means that, as permutations, $(c_1 \oplus c_2) \odot \mathit{swap}_+$ and $\mathit{swap}_+ \odot (c_2 \oplus c_1)$ are equal. And, of course, not all programs between the same types can be deformed into one another. The simplest example of inequivalent deformations are the two automorphisms of $1 + 1$, namely $\mathit{id}\leftrightarrow$ and swap_+.

While we will not make the details of the stretchable wires and slidable boxes formal, it is useful for intuition. One caveat though: some of the sliding and stretching needs to be done in spaces of higher dimension than 3 to have "enough room" to move things along without collision or over-stretching wires. That, unfortunately, means that some equivalences are harder to grasp. Luckily, most equivalences only need 3 dimensions.

Our reversible language of type isomorphisms and equivalences between them has a strong connection to *univalent universes* in HoTT [44]. Based on this connection, we refer to the types as being at level-0, to the equivalences between types (i.e., the combinators of Sec. 3) as being at level-1, and to the equivalences between equivalences of types (i.e., the combinators discussed in this section) as being at level-2.

4.1 A Model of Equivalences between Type Equivalences

Previously we saw how we could take the proof terms of commutative semiring equivalences as our starting point for Π. What we need now is to understand how *proofs* of algebraic identities should be considered equivalent. Classical algebra does not help, as proofs are not considered first-class citizens. However, another route is available to us: since the work of Hofmann and Streicher [45], we know that one can model types as *groupoids*. The additional structure comes from explicitly modeling the "identity types": instead of regarding all terms which witness the equality of (say) a and b of type A as being indistinguishable, we posit that there may in fact be many. This consequences of this one decision are enough to show that types can be modeled by groupoids.

Thus, rather than looking at (untyped) commutative semirings, we should look at a *typed* version. This process frequently goes by the moniker of "categorification." We want a categorical algebra, where the basic objects are groupoids (to model our types), and where there is a natural notion of $+$ and $*$. At first, we hit what seems like a serious stumbling block: the category of all groupoids, Groupoid, have neither co-products nor products. However, we don't want to work internally in Groupoid– we want operations *on* groupoids. In other words, we want something akin to symmetric monoidal categories, but with two interacting monoidal structures. Luckily, this already exists: the categorical analog to (commutative) semirings are (symmetric) Rig Categories [46, 47]. This straightforwardly generalizes to symmetric Rig Groupoids.

Let $c_1 : t_1 \leftrightarrow t_2$, $c_2 : t_3 \leftrightarrow t_4$, $c_3 : t_1 \leftrightarrow t_2$, and $c_4 : t_3 \leftrightarrow t_4$.
Let $a_1 : t_5 \leftrightarrow t_1$, $a_2 : t_6 \leftrightarrow t_2$, $a_3 : t_1 \leftrightarrow t_3$, and $a_4 : t_2 \leftrightarrow t_4$.

$$\frac{c_1 \Leftrightarrow c_3 \quad c_2 \Leftrightarrow c_4}{c_1 \oplus c_2 \Leftrightarrow c_3 \oplus c_4} \qquad \frac{c_1 \Leftrightarrow c_3 \quad c_2 \Leftrightarrow c_4}{c_1 \otimes c_2 \Leftrightarrow c_3 \otimes c_4}$$

$$(a_1 \odot a_3) \oplus (a_2 \odot a_4) \Leftrightarrow (a_1 \oplus a_2) \odot (a_3 \oplus a_4)$$

$$(a_1 \odot a_3) \otimes (a_2 \odot a_4) \Leftrightarrow (a_1 \otimes a_2) \odot (a_3 \otimes a_4)$$

Figure 6: Signatures of level-2 Π-combinators: functors

How does this help? Coherence conditions! Symmetric monoidal categories, to start somewhere simple, do not just introduce natural transformations like the associator α and the left and right unitors (λ and ρ respectively), but also coherence conditions that these must satisfy. Looking, for example, at just the additive fragment of Π (i.e. with just 0, 1 and + for the types, $\odot$ and $\oplus$ as combinators, and only the terms so expressible), the sub-language would correspond, denotationally, to exactly (non-empty) symmetric monoidal groupoids. And what these possess are exactly some *equations between equations* as commutative diagrams. Transporting these coherence conditions, for example those that express that various transformations are *natural* to Π, gives a list of equations between Π programs. Furthermore, all the natural transformations that arise are in fact natural *isomorphisms* – and thus reversible.

We can then proceed to prove that every one of the coherence conditions involved in defining a symmetric Rig Groupoid holds for the groupoid interpretation of types [32]. This is somewhat tedious given the sheer number of these, but when properly formulated, relatively straightforward, but see below for comments on some tricky cases.

But why are these particular coherence laws? Are they all necessary? Conversely are they, in some appropriate sense, sufficient? This is the so-called *coherence problem*. Mac Lane, in his farewell address as President of the American Mathematical Society [48] gives a good introduction and overview of such problems. A more modern interpretation (which can nevertheless be read into Mac Lane's own exposition) would read as follows: given a set of equalities on abstract words, regarded as a rewrite system, and two means of rewriting a word in that language to another, is there some suitable notion of canonical form that expresses the essential uniqueness of the non-trivial rewrites? Note how this word-and-rewrite problem is essentially independent of the eventual interpretation. But one must take some care, as there are obvious degenerate cases (involving "trivial" equations involving 0 or 1) which lead to non-uniqueness. The landmark results, first by Kelly-Mac Lane [49] for closed symmetric monoidal categories, then (independently) Laplaza and Kelly [46, 47] for symmetric Rig Categories, is that indeed there are sound and complete coherence conditions that insure that all the "obvious" equalities between different abstract words in these systems give rise to commutative diagrams. The "obvious" equalities come from *syzygies* or *critical pairs* of the system of equations. The problem is far from trivial — Fiore et al. [50] document some publications where the coherence set is in fact incorrect. They furthermore give a quite general algorithm to derive such coherence conditions.

4.2 A Language of Equivalences between Type Equivalences

As motivated in the previous section, the equivalences between type equivalences are perfectly modeled by the coherence conditions of weak Rig Groupoids. Syntactically, we take the easiest way there: simply make every coherence isomorphism into a programming construct. These constructs are collected in several figures (Fig. 7 to Fig. 15) and are discussed next.

Conveniently, the various coherence conditions can be naturally grouped into "related" laws. Each group basically captures the interactions between compositions of level-1 Π combinators.

Starting with the simplest constructions, the first two constructs in Fig. 6 are the level-2 analogs of +

Let $c_1 : t_1 \leftrightarrow t_2$, $c_2 : t_2 \leftrightarrow t_3$, and $c_3 : t_3 \leftrightarrow t_4$:

$$
\begin{aligned}
c_1 \odot (c_2 \odot c_3) &\Leftrightarrow (c_1 \odot c_2) \odot c_3 \\
(c_1 \oplus (c_2 \oplus c_3)) \odot \mathit{assocl}_+ &\Leftrightarrow \mathit{assocl}_+ \odot ((c_1 \oplus c_2) \oplus c_3) \\
(c_1 \otimes (c_2 \otimes c_3)) \odot \mathit{assocl}_\times &\Leftrightarrow \mathit{assocl}_\times \odot ((c_1 \otimes c_2) \otimes c_3) \\
((c_1 \oplus c_2) \oplus c_3) \odot \mathit{assocr}_+ &\Leftrightarrow \mathit{assocr}_+ \odot (c_1 \oplus (c_2 \oplus c_3)) \\
((c_1 \otimes c_2) \otimes c_3) \odot \mathit{assocr}_\times &\Leftrightarrow \mathit{assocr}_\times \odot (c_1 \otimes (c_2 \otimes c_3)) \\
\mathit{assocr}_+ \odot \mathit{assocr}_+ &\Leftrightarrow ((\mathit{assocr}_+ \oplus \mathit{id}\leftrightarrow) \odot \mathit{assocr}_+) \odot (\mathit{id}\leftrightarrow \oplus \mathit{assocr}_+) \\
\mathit{assocr}_\times \odot \mathit{assocr}_\times &\Leftrightarrow ((\mathit{assocr}_\times \otimes \mathit{id}\leftrightarrow) \odot \mathit{assocr}_\times) \odot (\mathit{id}\leftrightarrow \otimes \mathit{assocr}_\times)
\end{aligned}
$$

Figure 7: Signatures of level-2 Π-combinators: associativity

Let $c_1 : t_1 \leftrightarrow t_2$, $c_2 : t_3 \leftrightarrow t_4$, and $c_3 : t_5 \leftrightarrow t_6$:

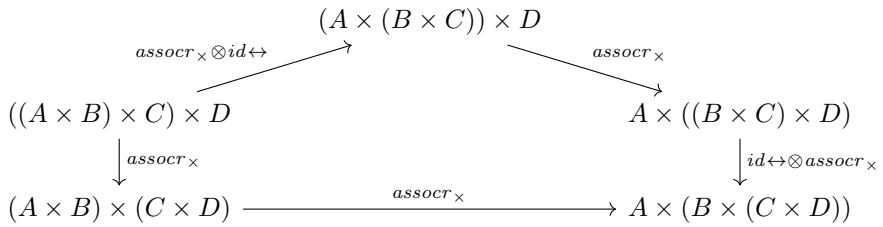
$$
\begin{aligned}
((c_1 \oplus c_2) \otimes c_3) \odot \mathit{dist} &\Leftrightarrow \mathit{dist} \odot ((c_1 \otimes c_3) \oplus (c_2 \otimes c_3)) \\
(c_1 \otimes (c_2 \oplus c_3)) \odot \mathit{distl} &\Leftrightarrow \mathit{distl} \odot ((c_1 \otimes c_2) \oplus (c_1 \otimes c_3)) \\
((c_1 \otimes c_3) \oplus (c_2 \otimes c_3)) \odot \mathit{factor} &\Leftrightarrow \mathit{factor} \odot ((c_1 \oplus c_2) \otimes c_3) \\
((c_1 \otimes c_2) \oplus (c_1 \otimes c_3)) \odot \mathit{factorl} &\Leftrightarrow \mathit{factorl} \odot (c_1 \otimes (c_2 \oplus c_3))
\end{aligned}
$$

Figure 8: Signatures of level-2 Π-combinators: distributivity and factoring

and $*$, which respectively model level-1 choice composition and parallel composition (of equivalences). These allow us to "build up" larger equivalences from smaller ones. The next two express that both of these composition operators distribute over sequential composition $\odot$ (and vice versa).

The constructs in Fig. 7 capture the informal idea that all the different ways of associating programs are equivalent. The first says that sequential composition itself ($\odot$) is associative. The next 4 capture how the $\oplus$ and $\otimes$ combinators "commute" with re-association. In other words, it expresses that the type-level associativity of $+$ is properly reflected by the properties of $\oplus$. The last two equivalences show how composition of associativity combinators interact together.

The bottom line in Fig. 7 is actually a linear restatement of the famous "pentagon diagram" stating a particular coherence condition for monoidal categories [49]. To make the relation between Π as a language and the language of category theory, the figure below displays the same morphism but in categorical terms.



The constructs in Fig. 8 are the basic coherence for *dist*, *distl*, *factor* and *factorl*: the type-level distribution and factoring has to commute with the level-1 $\oplus$ and $\otimes$.

The constructs in Fig. 9 express various properties of composition. The first two says that $\mathit{id}\leftrightarrow$ is a left and right identity for sequential composition. The next two say that all programs are reversible, both on the left and the right: running c and then its reverse $(!\, c)$ is equivalent to the identity, and the same for doing $!\, c$ first then c. The last line say that there is an identity level-2 combinator, a sequential composition, and that level-2 equivalence respects level-1 sequential composition $\odot$.

Let $c_0, c_1, c_2, c_3 : t_1 \leftrightarrow t_2$ and $c_4, c_5 : t_3 \leftrightarrow t_4$:

$$id\!\leftrightarrow \odot c_0 \Leftrightarrow c_0 \quad c_0 \odot id\!\leftrightarrow \Leftrightarrow c_0 \quad c_0 \odot\, !\, c_0 \Leftrightarrow id\!\leftrightarrow \quad !\, c_0 \odot c_0 \Leftrightarrow id\!\leftrightarrow$$

$$id\!\leftrightarrow \oplus id\!\leftrightarrow \Leftrightarrow id\!\leftrightarrow \qquad id\!\leftrightarrow \otimes id\!\leftrightarrow \Leftrightarrow id\!\leftrightarrow$$

$$c_0 \Leftrightarrow c_0 \qquad \frac{c_1 \Leftrightarrow c_2 \quad c_2 \Leftrightarrow c_3}{c_1 \Leftrightarrow c_3} \qquad \frac{c_1 \Leftrightarrow c_4 \quad c_2 \Leftrightarrow c_5}{c_1 \odot c_2 \Leftrightarrow c_4 \odot c_5}$$

Figure 9: Signatures of level-2 Π-combinators: identity and composition

Let $c_0 : 0 \leftrightarrow 0$, $c_1 : 1 \leftrightarrow 1$, and $c_3 : t_1 \leftrightarrow t_2$:

$$unite_+ l \odot c_3 \Leftrightarrow (c_0 \oplus c_3) \odot unite_+ l \qquad uniti_+ l \odot (c_0 \oplus c_3) \Leftrightarrow c_3 \odot uniti_+ l$$

$$unite_+ r \odot c_3 \Leftrightarrow (c_3 \oplus c_0) \odot unite_+ r \qquad uniti_+ r \odot (c_3 \oplus c_0) \Leftrightarrow c_3 \odot uniti_+ r$$

$$unite_\times l \odot c_3 \Leftrightarrow (c_1 \otimes c_3) \odot unite_\times l \qquad uniti_\times l \odot (c_1 \otimes c_3) \Leftrightarrow c_3 \odot uniti_+ l$$

$$unite_\times r \odot c_3 \Leftrightarrow (c_3 \otimes c_1) \odot unite_\times r \qquad uniti_\times r \odot (c_3 \otimes c_1) \Leftrightarrow c_3 \odot uniti_\times r$$

$$unite_\times l \Leftrightarrow distl \odot (unite_\times l \oplus unite_\times l)$$

$$unite_+ l \Leftrightarrow swap_+ \odot unite_+ r \qquad unite_\times l \Leftrightarrow swap_\times \odot unite_\times r$$

Figure 10: Signatures of level-2 Π-combinators: unit

The constructs in Fig. 10 may at first blush look similarly straightforward, but deserve some pause. One obvious question: What is the point of $c_0 : 0 \leftrightarrow 0$, isn't that just the identity combinator $id\!\leftrightarrow$ for $A = 0$ (as defined in Fig. 1)? Operationally, c_0 is indeed indistinguishable from $id\!\leftrightarrow$. However, there are multiple syntactic ways of writing down combinators of type $0 \leftrightarrow 0$, and the first combinator in Fig. 10 applies to all of them uniformly. This is another subtle aspect of coherence: all reasoning must be valid for all possible models, not just the one we have in mind. So even though operational reasoning may suggest that some relations *may* be true between combinators, it can also mislead. The same reasoning applies to $c_1 : 1 \leftrightarrow 1$. The first 8 combinators can then be read as basic coherence for unit introduction and elimination, in both additive and multiplicative cases.

The last two capture another simple idea, related to swapping: eliminating a unit on the left is the same as first swapping then eliminating on the right (both additively and multiplicatively). As a side note, these are not related to *commutativity*, but rather come from one of the simplest coherence condition for braided monoidal categories. In other words, it reflects the idempotence of $swap_+$ and $swap_\times$ rather than the commutativity of $\oplus$ and $\otimes$.

The first two equivalences in Fig. 11 reflect the basic coherence between level-0 swapping and the level-1 combinator actions. The next four arise because of interactions between (additive and multiplicative) level-1 associativity and swapping. In other words, they arise as critical pairs. For example, the first expresses that the two ways of going from $(A \oplus B) \oplus C$ to $B \oplus (C \oplus A)$ are equivalent, with the second saying that the reverse (i.e. the results of applying $!$) also gives equivalent programs. The last two say the same but for the multiplicative structure.

The constructs in Fig. 12 express how unit elimination "in the middle" can be expressed either as operating on the right or, (after re-association) on the left.

The constructs in Fig. 13 are significantly more subtle, as they deal with combinators involving 0, aka an impossibility. For example,

$$(c \otimes id\!\leftrightarrow_0) \odot absorbl \Leftrightarrow absorbl \odot id\!\leftrightarrow_0$$

(where we have explicitly annotated the types of $id\!\leftrightarrow$ for increased clarity) tells us that of the two ways

Let $c_1 : t_1 \leftrightarrow t_2$ and $c_2 : t_3 \leftrightarrow t_4$:

$$swap_+ \odot (c_1 \oplus c_2) \Leftrightarrow (c_2 \oplus c_1) \odot swap_+ \quad swap_\times \odot (c_1 \otimes c_2) \Leftrightarrow (c_2 \otimes c_1) \odot swap_\times$$
$$(assocr_+ \odot swap_+) \odot assocr_+ \Leftrightarrow ((swap_+ \oplus id\leftrightarrow) \odot assocr_+) \odot (id\leftrightarrow \oplus swap_+)$$
$$(assocl_+ \odot swap_+) \odot assocl_+ \Leftrightarrow ((id\leftrightarrow \oplus swap_+) \odot assocl_+) \odot (swap_+ \oplus id\leftrightarrow)$$
$$(assocr_\times \odot swap_\times) \odot assocr_\times \Leftrightarrow ((swap_\times \otimes id\leftrightarrow) \odot assocr_\times) \odot (id\leftrightarrow \otimes swap_\times)$$
$$(assocl_\times \odot swap_\times) \odot assocl_\times \Leftrightarrow ((id\leftrightarrow \otimes swap_\times) \odot assocl_\times) \odot (swap_\times \otimes id\leftrightarrow)$$

Figure 11: Signatures of level-2 Π-combinators: commutativity and associativity

$$unite_+ r \oplus id\leftrightarrow \quad \Leftrightarrow \quad assocr_+ \odot (id\leftrightarrow \oplus unite_+ l)$$
$$unite_\times r \otimes id\leftrightarrow \quad \Leftrightarrow \quad assocr_\times \odot (id\leftrightarrow \otimes unite_\times l)$$

Figure 12: Signatures of level-2 Π-combinators: unit and associativity

of transforming from $t_1 * \ 0$ to 0, namely first doing some arbitrary transformation c from t_1 to t_2 and (in parallel) leaving 0 alone then eliminating 0, or first eliminating 0 then doing the identity (at 0), are equivalent. This is the "naturality" of *absorbl*. One item to note is the fact that this combinator is not irreducible, as the $id\leftrightarrow$ on the right can be eliminated. But that is actually a property visible at an even higher level (which we will not touch in this paper). The next 3 are similarly expressing the naturality of *absorbr*, *factorzl* and *factorzr*.

The next combinator, $absorbr \Leftrightarrow absorbl$, is particularly fascinating: while it says something simple — that the two obvious ways of transforming $0 * 0$ into 0, namely absorbing either the left or right 0 — it implies something subtle. A straightforward proof of *absorbl* which proceeds by saying that $0 * t$ cannot be inhabited because the first member of the pair cannot, is not in fact equivalent to *absorbr* on $0 * 0$. However, if we instead define *absorbl* to "transport" the putative impossible first member of the pair to its (equally impossible) output, then these do form equivalent pairs. The next few in Fig. 13 also express how *absorbr* and *absorbl* interact with other combinators. As seen previously, all of these arise as critical pairs. What is much more subtle here is that the types involved often are asymmetric: they do not have the same occurrences on the left and right. Such cases are particularly troublesome for finding normal forms. Laplaza [46] certainly comments on this, but in mostly terse and technical terms. Blute et al. [42] offer much more intuitive explanations.

The constructs in Fig. 14 and Fig. 15 relating associativity and distributivity, and commutativity and distributivity, have more in common with previous sets of combinators. They do arise from non-trivial critical pairs of different ways of going between equivalent types. The last one of Fig. 14 is particularly daunting, involving a sequence of 3 combinators on the left and 6 on the right.

4.3 Operational Semantics

There are two different interpretations for an operational semantics for the language of equivalences:

1. Mimicking closely the one in Sec. 3.3, and thus finding explicit homotopies between the functions induced by the operational semantics of the level-1 combinators.

2. Treating things more syntactically, and interpreting the combinators as program transformations.

A previous paper [32] explores the first interpretation in depth. There one can find a definition of "equivalences of equivalences", which as the base of that interpretation.

Let $c : t_1 \leftrightarrow t_2$:

$$(c \otimes id\!\leftrightarrow) \odot absorbl \Leftrightarrow absorbl \odot id\!\leftrightarrow \quad (id\!\leftrightarrow \otimes c) \odot absorbr \Leftrightarrow absorbr \odot id\!\leftrightarrow$$
$$id\!\leftrightarrow \odot factorzl \Leftrightarrow factorzl \odot (id\!\leftrightarrow \otimes c) \quad id\!\leftrightarrow \odot factorzr \Leftrightarrow factorzr \odot (c \otimes id\!\leftrightarrow)$$
$$absorbr \Leftrightarrow absorbl$$
$$absorbr \Leftrightarrow (distl \odot (absorbr \oplus absorbr)) \odot unite_+ l$$
$$unite_\times r \Leftrightarrow absorbr \qquad absorbl \Leftrightarrow swap_\times \odot absorbr$$
$$absorbr \Leftrightarrow (assocl_\times \odot (absorbr \otimes id\!\leftrightarrow)) \odot absorbr$$
$$(id\!\leftrightarrow \otimes absorbr) \odot absorbl \Leftrightarrow (assocl_\times \odot (absorbl \otimes id\!\leftrightarrow)) \odot absorbr$$
$$id\!\leftrightarrow \otimes unite_+ l \Leftrightarrow (distl \odot (absorbl \oplus id\!\leftrightarrow)) \odot unite_+ l$$

Figure 13: Signatures of level-2 Π-combinators: zero

$$((assocl_+ \otimes id\!\leftrightarrow) \odot dist) \odot (dist \oplus id\!\leftrightarrow) \Leftrightarrow (dist \odot (id\!\leftrightarrow \oplus dist)) \odot assocl_+$$
$$assocl_\times \odot distl \Leftrightarrow ((id\!\leftrightarrow \otimes distl) \odot distl) \odot (assocl_\times \oplus assocl_\times)$$
$$\begin{aligned}(distl \odot (dist \oplus dist)) \odot assocl_+ \quad \Leftrightarrow \quad & dist \odot (distl \oplus distl) \odot assocl_+ \odot \\ & (assocr_+ \oplus id\!\leftrightarrow) \odot \\ & ((id\!\leftrightarrow \oplus swap_+) \oplus id\!\leftrightarrow) \odot \\ & (assocl_+ \oplus id\!\leftrightarrow)\end{aligned}$$

Figure 14: Signatures of level-2 Π-combinators: associativity and distributivity

Here we will focus instead of the syntactic interpretation as program transformers. This results in a function:

$$eval_1 \; : \; \{t_1\, t_2 \; : \; U\}\, \{c_1\, c_2 \; : \; t_1 \; \leftrightarrow \; t_2\}\, (ce \; c_1 \Leftrightarrow \; c_2) \to (t_1 \; \leftrightarrow \; t_2)$$

This function is "deeply dependent": given the type of the rewrite ce to apply, both the input c_1 and output c_2 are almost entirely determined! Let us take for example the second combinator in Fig. 8:

$$(c_1 \otimes (c_2 \oplus c_3)) \odot distl \Leftrightarrow distl \odot ((c_1 \otimes c_2) \oplus (c_1 \otimes c_3))$$

which we can name distl⇔l. Interpreting this as a rewrite from the program on the left to the one on the right requires "pattern matching" on the left structure which contains 3 arbitrary combinators, from which we can *reconstruct* the program on the right. Rewrites such as distl⇔l are one-step rewrites, in the same way that *distl* is a constant of the base term language of Π. There is one additional wrinkle. There is naturally an opposite combinator, which interprets the above from right to left; let us call it distl⇔r. It would appear to require *non-linear pattern-matching* since the right-hand-side contains c_1 twice. That is however not the case! The definition of distl⇔r has 5 implicit arguments, 3 of which are c_1, c_2, and c_3, which then completely force the "shape" of the overall pattern. Thus the mere mention of distl⇔r is enough to resolve the apparent use of a non-linear pattern. This is why $eval_1$ was called "deeply dependent" above: once the name of the combinator is given, the rest follows.

If all expressible transformations were single-step only, this would hardly justify calling this an "operational semantics," as we would hardly have a programming language. However, level-2 of Π has combinators as well: two are in Fig. 6 and two are in Fig. 9. The most interesting one is "sequential composition," which is the middle one at the bottom of Fig. 9. Since ⇔ represents an equivalence, sequential composition in

$$(id\leftrightarrow \otimes swap_+) \odot distl \quad \Leftrightarrow \quad distl \odot swap_+$$
$$dist \odot (swap_\times \oplus swap_\times) \quad \Leftrightarrow \quad swap_\times \odot distl$$

Figure 15: Signatures of level-2 Π-combinators: commutativity and distributivity

this context is the same as transitivity of equivalences, as thus we have chosen to name this trans⇔. When evaluating trans⇔, we could cheat: we know that the eventual answer must be, and we could just return that. But this is not operational in any real sense, as that skips over the intermediate steps. We would like to be able to "trace" the rewrite. Thus the evaluation of trans⇔ r_0 r_1 where $r_0 : c_0 \leftrightarrow c_1$ and $r_1 : c_1 \leftrightarrow c_2$ should apply $eval_1$ to both r_0 and r_1. Furthermore, after applying r_0, we should be able to witness that the result is indeed c_1, so that we may continue. This last requirement forces us to define a new function, mutually recursively with $eval_1$, for this task:

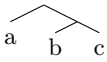
$$exact \; : \; \{t_1\, t_2 \; : \; U\}\, \{c_1\, c_2 \; : \; t_1 \; \leftrightarrow \; t_2\}\, (ce \; c_1 \Leftrightarrow \; c_2) \to eval_1\, ce \equiv c_2$$

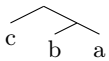
If we are careful in our construction of $eval_1$, the definition of *exact* is quite straightforward, i.e. almost all cases are immediately provable by reflexivity.

This then lets us define the trans⇔ r_0 r_1 case properly: we first evaluate r_0 and get a result combinator, witness that this result type is indeed exactly what we expect, and proceed to evaluate r_1 where we specify that the r_1's left-hand side must be $eval_1\ r_0$; we can use the Agda keyword rewrite to make this match c_2 "on the nose" (otherwise the call would be ill-typed). This then forces us to use rewrite also in the implementation of the trans⇔ case in *exact*.

The other three combinators are much simpler, as simple recursive calls are sufficient.

4.4 Example

We can now illustrate how this all works with a small example. Consider a circuit that takes an input type consisting of three values 

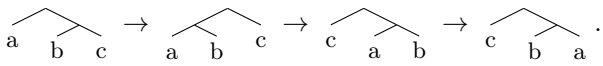
and swaps the leftmost value with the rightmost value to produce 

. We can implement two such circuits using our Agda library for Π:

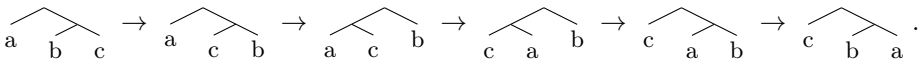
```
swap-fl1 swap-fl2 : {a b c : U} → PLUS a (PLUS b c) ↔ PLUS c (PLUS b a)
swap-fl1 = assocl+ ⊙ swap+ ⊙ (id↔ ⊕ swap+)

swap-fl2 = (id↔ ⊕ swap+) ⊙
           assocl+ ⊙
           (swap+ ⊕ id↔) ⊙
           assocr+ ⊙
           (id↔ ⊕ swap+)
```

The first implementation rewrites the incoming values as follows:



The second implementation rewrites the incoming values as follows:



The two circuits are extensionally equal. Using the level-2 isomorphisms we can *explicitly* construct a sequence of rewriting steps that transforms the second circuit to the first.

We write such proofs in an equational style: in the left column, we have the current combinator which is equivalent to the first one, and in the right column, the justification for that equivalence. The joining combinator is syntactic sugar for trans⇔. The transformation could be written (using trans⇔) by just giving all the pieces in the right hand column — but such transformations are very hard for humans to understand and follow.

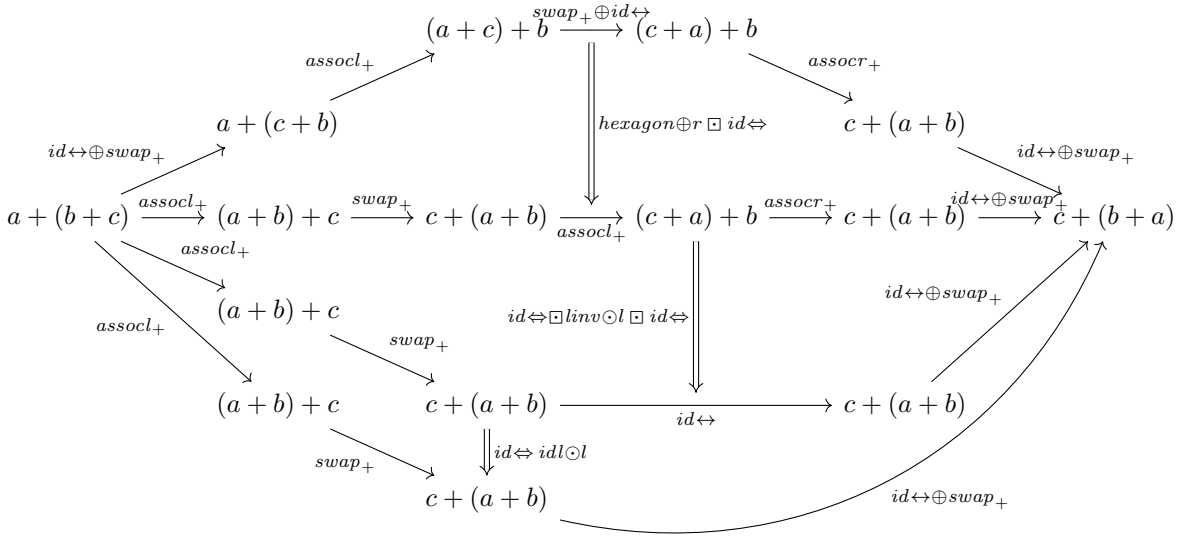
The proof can be read as follows: the first three lines "refocus" from a right-associated isomorphism onto the (left-associated) composition of the first 3 isomorphisms; then apply a complex rewrite on these (the "hexagon" coherence condition of symmetric braided monoidal categories); this exposes two inverse combinators next to each other — so we have to refocus on these to eliminate them; we finally re-associate to get the result.

```
swap-fl2⇔swap-fl1 : {a b c : U} → swap-fl2 {a} {b} {c} ⇔ swap-fl1
swap-fl2⇔swap-fl1 =
  ((id↔ ⊕ swap₊) ⊙ assocl₊ ⊙ (swap₊ ⊕ id↔) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))     ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
  ((id↔ ⊕ swap₊) ⊙ (assocl₊ ⊙ (swap₊ ⊕ id↔)) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))   ⇔⟨ assoc⊙l ⟩
  (((id↔ ⊕ swap₊) ⊙ assocl₊ ⊙ (swap₊ ⊕ id↔)) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))   ⇔⟨ assoc⊙l ⊡ id⇔ ⟩
  ((((id↔ ⊕ swap₊) ⊙ assocl₊) ⊙ (swap₊ ⊕ id↔)) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊)) ⇔⟨ hexagonl⊕r ⊡ id⇔ ⟩
  (((assocl₊ ⊙ swap₊) ⊙ assocl₊) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))               ⇔⟨ assoc⊙r ⟩
  ((assocl₊ ⊙ swap₊) ⊙ assocl₊ ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))                 ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
  ((assocl₊ ⊙ swap₊) ⊙ (assocl₊ ⊙ assocr₊) ⊙ (id↔ ⊕ swap₊))               ⇔⟨ id⇔ ⊡ (linv⊙l ⊡ id⇔) ⟩
  ((assocl₊ ⊙ swap₊) ⊙ id↔ ⊙ (id↔ ⊕ swap₊))                              ⇔⟨ id⇔ ⊡ idl⊙l ⟩
  ((assocl₊ ⊙ swap₊) ⊙ (id↔ ⊕ swap₊))                                    ⇔⟨ assoc⊙r ⟩
  ((assocl₊ ⊙ swap₊ ⊙ (id↔ ⊕ swap₊)) □)
```

4.5 Internal Language

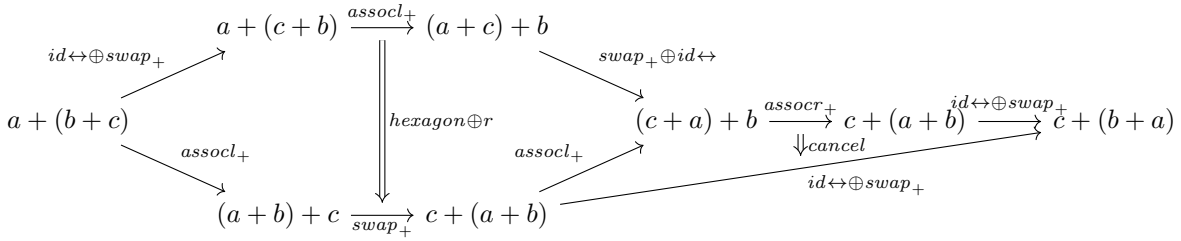
Recalling that the λ-calculus arises as the internal language of Cartesian Closed Categories (Elliott [51] gives a particularly readable account of this), we can think of Π in similar terms, but for symmetric Rig Groupoids instead. For example, we can ask what does the derivation in Sec. 4.4 represent? It is actually a "linear" representation of a 2-categorial commutative diagram! In fact, it is a painfully verbose version thereof, as it includes many *refocusing* steps because our language does not build associativity into its syntax. Categorical diagrams usually do. Thus if we rewrite the example in diagrammatic form, eliding all uses of associativity,

but keeping explicit uses of identity transformations, we get that swap-fl2⇔swap-fl1 represents



For some, the above diagram will be clearer — it is only three layers high rather than nine! Others will prefer the more programmatic feel of the original definition.

We would be remiss in letting the reader believe that the above is "the" categorical diagram that would be found in categorical textbooks. Rather, congruence would be used to elide the $id \Leftrightarrow$. Furthermore, the various arrows would also be named differently — our $assocl_+$ is often named α, $assocr_+$ is α^{-1}, $swap_+$ is B (always with subscripts). And the two steps needed to remove inverses (i.e. first cancelling inverse arrows, then removing the resulting identity arrow "in context") are often combined into one. Here we'll simply name this operation *cancel*, which could be programmed as a defined function over Π level-2. The result would then be the much simpler



In other words, each (non-refocusing) line of the proof of swap-fl2⇔swap-fl1 is a complete path from left to right in each diagram above, and the annotation on the right-hand-side becomes the natural transformation (denoted by vertical ⇒) justifying the move to the next line. The first diagram uses lines $1, 4, 7, 8$ in full; the second diagram collapses 7 and 8 into one, as well as not duplicating parts which are related by $id \Leftrightarrow$.

5 Further Thoughts and Conclusions

We conclude with a collection of open problems and avenues for further research.

5.1 Richer Data: Infinite Sets and Topological Spaces

The three languages we discussed only deal with the finite spaces built from 0, 1, sums, and products. Programming practice, logic, and mathematics all deal with richer spaces including inductive types (e.g., the natural numbers, sequences, and trees), functions, and graphs. Extending Π to such domains is possible but only after one refines the notions of reversibility and conservation of information. One approach is to use *partial isomorphisms* that may be undefined on such inputs [52, 30]. Another more speculative approach is to build such spaces, topologically, based on novel type constructions such as negative, fractional, or even imaginary types [53, 54].

5.2 Information Effects

A computational model that enforces the principle of conservation of information is arguably *richer* than a conventional model that cannot even express the notion of information. Practically the conventional model is easily recovered by simply adding constructs that intentionally and explicitly create or erase information. Such constructs allow one to recover the classical perspective with the added advantage that it is possible to reason about such creation and erasure of information using type and effect systems, monads, or arrows [52, 55].

An interesting application of such an idea is in the field of *information-flow security*. To make this idea concrete, consider a tiny 2-bit password = `"10"` and the associated password checker:

```
check-password (guess) =
  guess == "10"
```

One can ask how much information is leaked by this program assuming the attacker has no prior knowledge except that the password is 2 bits, i.e., the four possible 2-bits are equally likely. If the attacker guesses `"10"` (with probability 1/4) the password (2 bits) is leaked. If the attacker guesses one of the other choices (with probability 3/4) the number of possibilities is reduced from 4 to 3, i.e., the attacker learns $\log 4 - \log 3$ bits of information. So in general the attacker learns:

$$\begin{array}{rl} & 1/4 * 2 + 3/4(\log 4 - \log 3) \\ = & 1/4 \log 4 + 3/4 \log 4/3 \\ = & -1/4 \log 1/4 - 3/4 \log 3/4 \\ \sim & 0.8 \text{ bits in the first probe} \end{array}$$

This is a significant amount of information. But of course this is only because the password is so short: if the password was 8 restricted ASCII characters (6 bits), the attacker would only learn 0.00001 bits in the first probe.

An alternative formulation of the problem is to view the input as a random variable with 4 possibilities and a uniform distribution (i.e., with 2 bits of information) and the output as another random variable with 4 possibilities but with the distribution $\{(\mathit{True}, 1/4), (\mathit{False}, 3/4)\}$ which contains 0.8 bits of information. Thus 2 input bits of information were given to the password checker and only 0.8 were produced. Where did the 1.2 bits of information go? By the Landauer Principle, these 1.2 bits must be accounted by an *implicit erasure* operation in the program. By writing the password checker in an extension of Π, the erasure construct becomes explicit and the information leak becomes exposed in the syntactic structure of the program [52].

5.3 Theseus and Quantum Control

The Π family of languages semantically captures the principles of reversibility and conservation of information. As a programming language it has some mixed properties: small programs are relatively easy to write; for some special classes of programs, it is even possible to define a methodology to write large Π programs, including a meta-circular interpreter for Π [56]. In general, however, the point-free style of combinators used in Π becomes awkward and a new approach appears more suitable. To that end, we note that Π encodes the

```
f :: Either Int Int -> a
f (Left 0)     = undefined
f (Left (n+1)) = undefined
f (Right n)    = undefined
```

Figure 16: A skeleton

```
g :: (Bool,Int) -> a
g (False,n)  = undefined
g (True,0)   = undefined
g (True,n+1) = undefined
```

Figure 17: Another skeleton

```
h :: Either Int Int <-> (Bool,Int)
h (Left 0)     = (True,0)
h (Left (n+1)) = (False,n)
h (Right n)    = (True,n+1)
```

Figure 18: An isomorphism

most elementary control structure in a programming language– which is the ability to conditionally execute one of several possible code fragments– using combinators. Expressing such an abstraction using combinators or even predicates and nested **if**-expressions makes it difficult for both humans and compilers to write, understand, and reason about the control flow structure of the program. Instead, in modern functional languages, this control flow paradigm is elegantly expressed using *pattern-matching*. This approach yields code that is not only more concise and readable but also enables the compiler to easily verify two crucial properties: (i) non-overlapping patterns and (ii) exhaustive coverage of a datatype using a collection of patterns. Indeed most compilers for functional languages perform these checks, warning the user when they are violated. At a more fundamental level, e.g., in type theories and proof assistants, these properties are actually necessary for correct reasoning about programs. Our insight is that these properties, perhaps surprisingly, are sufficient to produce a simple and intuitive first-order reversible programming language which we call *Theseus*.

We provide a small illustrative example, written in a Haskell-like syntax. Fig. 16 gives the skeleton of a function `f` that accepts a value of type `Either Int Int`; the patterns on the left-hand side exhaustively cover every possible incoming value and are non-overlapping. Similarly, Fig. 17 gives the skeleton for a function `g` that accepts a value of type `(Bool,Int)`; again the patterns on the left-hand side exhaustively cover every possible incoming value and are non-overlapping. Now we claim that since the types `Either Int Int` and `(Bool,Int)` are isomorphic, we can combine the patterns of `f` and `g` into *symmetric pattern-matching clauses* to produce a reversible function between the types `Either Int Int` and `(Bool,Int)`. Fig. 18 gives one such function; there, we suggestively use `<->` to indicate that the function can be executed in either direction. This reversible function is obtained by simply combining the non-overlapping exhaustive patterns on the two sides of a clause. In order to be well-formed in either direction, these clauses are subject to the constraint that each variable occurring on one side must occur exactly once on the other side (and with the same type). Thus it is acceptable to swap the second and third right-hand sides of `h` but not the first and second ones. With some additional work, it is possible to extend Theseus to a full-fledged reversible programming language [57]. With just one additional insight, Theseus can be extended with superpositions and becomes a quantum programming language [58].

5.4 Quantum Speed-up

A rather remarkable but somehow overlooked paper is "Quantum speedup and Categorical Distributivity" by Peter Hines [59]. Here he shows that the heart of Shor's algorithm can be reduced to an operation $!^N()$ (expressible in Π), which can be expressed, via a factorization, in an exponentially faster manner. The key to this efficient factorization is exactly the coherence conditions of Laplaza [46], which also feature prominently in our work. Proving his key Lemma 2 in Π could be quite instructive in revealing which level-2 combinators are crucial for this result.

5.5 Summary

The entire edifice of computer science including its mainstream models of computations, programming languages, and logics is founded on *classical physics.* While much of the world phenomena can be approximated with classical physics, we are reaching a revolutionary period of quantum technology that challenges many of the classical assumptions. It remains to be seen how computer science will adapt to this quantum revolution but we believe that additional physical principles inspired by quantum mechanics will have to be embraced in our computational thinking. This paper focused on one such principle — *conservation of information* — and explored some of its exciting implications to the field of computer science.

Acknowledgements

We would like to thank the numerous students and colleagues who participated in various aspects of this research and who provided valuable feedback and constructive criticism.

References

[1] Tommaso Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.

[2] Richard Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467–488, 1982.

[3] H. Leff and R. Rex. *Maxwell's Demon: Entropy, Information, Computing.* Princeton University Press, Princeton, NJ, 1990.

[4] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, July 1961.

[5] A. Bérut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483, 2012.

[6] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *2*, pages 230–265, 1936. Available from: `http://www.abelard.org/turpap2/tp2-ie.asp`.

[7] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951. (second printing, first appeared 1941).

[8] Claude Elwood Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656, 1948.

[9] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17:525–532, November 1973.

[10] C.H. Bennett and R. Landauer. The fundamental physical limits of computation. *Scientific American*, 253(1):48–56, 1985.

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

[12] C.H. Bennett. 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):501–510, 2003.

[13] Henry G. Baker. Nreversal of fortune - the thermodynamics of garbage collection. In *Proceedings of the International Workshop on Memory Management*, pages 507–524. Springer-Verlag, 1992.

[14] J. Baez and M. Stay. Physics, topology, logic and computation: a Rosetta Stone. *New Structures for Physics*, pages 95–172, 2011.

[15] Pasquale Malacaria and Fabrizio Smeraldi. The thermodynamics of confidentiality. In *CSF*, pages 280–290, 2012.

[16] Jacob D. Bekenstein. Universal upper bound on the entropy-to-energy ratio for bounded systems. *Phys. Rev. D*, 23:287–298, Jan 1981.

[17] Jean-Yves Girard. Truth, modality and intersubjectivity. *Mathematical. Structures in Comp. Sci.*, 17(6):1153–1167, December 2007.

[18] Henry G. Baker. Lively linear Lisp: — look ma, no garbage! —. *SIGPLAN Not.*, 27:89–98, August 1992.

[19] Asher Peres. Reversible logic and quantum computers. *Phys. Rev. A*, 32(6), Dec 1985.

[20] Michael P. Frank. *Reversibility for efficient computing*. PhD thesis, Massachusetts Institute of Technology, 1999.

[21] A. van Tonder. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, 2004.

[22] Werner E. Kluge. A reversible SE(M)CD machine. In *International Workshop on Implementation of Functional Languages*, pages 95–113. Springer-Verlag, 2000.

[23] Lorenz Huelsbergen. A logically reversible evaluator for the call-by-name lambda calculus. *InterJournal Complex Systems*, 46, 1996.

[24] V. Danos and J. Krivine. Reversible communicating systems. *Concurrency Theory*, pages 292–307, 2004.

[25] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *PEPM*, pages 144–153. ACM, 2007.

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

[27] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *MPC*, pages 289–313, 2004.

[28] Samson Abramsky. A structural approach to reversible computation. *Theor. Comput. Sci.*, 347:441–464, December 2005.

[29] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Reversible combinatory logic. *MSCS*, 16:621–637, August 2006.

[30] William J. Bowman, Roshan P. James, and Amr Sabry. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *Workshop on Reversible Computation*, 2011.

[31] Roshan P. James and Amr Sabry. Information effects. In *POPL*, pages 73–84. ACM, 2012.

[32] Jacques Carette and Amr Sabry. *ESOP 2016*, chapter Computing with Semirings and Weak Rig Groupoids, pages 123–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[33] Anthony J. G. Hey, editor. *Feynman and Computation: Exploring the Limits of Computers*. Perseus Books, Cambridge, MA, USA, 1999.

[34] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.

[35] Masahito Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *TLCA*. Springer-Verlag, 1997.

[36] Bart Desoete and Alexis De Vos. Feynman's reversible logic gates, implemented in silicon. In *Proceedings of the 6 th Advanced Training Course on Mixed Design of VLSI Circuits, Technical University Lodz, Napieralski A. (ed.), Krakow, juni 1999*, pages 497–502, 1999.

[37] Yvan Van Rentergem and Alexis De Vos. Optimal design of a reversible full adder. *IJUC*, 1:339–355, 2005.

[38] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[39] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. In *POPL*, pages 77–88. ACM, 2004.

[40] M. P. Fiore, R. Di Cosmo, and V. Balat. Remarks on isomorphisms in typed calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1-2):35–50, 2006.

[41] Ian Mackie. The geometry of interaction machine. In *POPL*, pages 198–208, 1995.

[42] R.F. Blute, J.R.B. Cockett, R.A.G. Seely, and T.H. Trimble. Natural deduction and coherence for weakly distributive categories. *Journal of Pure and Applied Algebra*, 113(3):229 – 296, 1996.

[43] Peter Selinger. A survey of graphical languages for monoidal categories. In Bob Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 289–355. Springer Berlin / Heidelberg, 2011.

[44] Jacques Carette, Chao-Hong Chen, Vikraman Choudhury, and Amr Sabry. From reversible programs to univalent universes and back. *Electronic Notes in Theoretical Computer Science*, 336:5 – 25, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

[45] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Venice Festschrift*, pages 83–111, 1996.

[46] Miguel L. Laplaza. Coherence for distributivity. In G.M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane, editors, *Coherence in Categories*, volume 281 of *Lecture Notes in Mathematics*, pages 29–65. Springer Verlag, Berlin, 1972.

[47] G. M. Kelly. Coherence theorems for lax algebras and for distributive laws. In Gregory M. Kelly, editor, *Category Seminar*, pages 281–375, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

[48] Saunders Mac Lane. Topology and logic as a source of algebra. *Bull. Amer. Math. Soc.*, 82:1–40, 1976.

[49] G.M. Kelly and Saunders Mac Lane. Coherence in closed categories. *Journal of Pure and Applied Algebra*, 1(1):97 – 140, 1971.

[50] Thomas M. Fiore, Po Hu, and Igor Kriz. Laplaza sets, or how to select coherence diagrams for pseudo algebras. *Advances in Mathematics*, 218(6):1705 – 1722, 2008.

[51] Conal Elliott. Compiling to categories. *Proc. ACM Program. Lang.*, 1(ICFP), September 2017.

[52] Roshan P. James and Amr Sabry. Information effects. In *POPL*, 2012.

[53] A. Blass. Seven trees in one. *Journal of Pure and Applied Algebra*, 103(1-21), 1995.

[54] Roshan P. James. The Computational Content of Isomorphisms. In *PhD Thesis, Indiana University*, 2013.

[55] C. Heunen, R. Kaarsgaard, and M. Karvonen. Reversible effects as inverse arrows. In *MFPS*, 2018.

[56] Roshan P. James and Amr Sabry. Isomorphic Interpreters from Small-Step Abstract Machines. In *Reversible Computation*, 2012.

[57] Roshan P. James and Amr Sabry. Theseus: A high-level language for reversible computation. In *Reversible Computation*, 2014. Booklet of work-in-progress and short reports.

[58] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. From symmetric pattern-matching to quantum control. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 348–364, Cham, 2018. Springer International Publishing.

[59] Peter Hines. Quantum speedup and categorical distributivity. In *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*, pages 122–138. Springer, 2013.