Counting on Type Isomorphisms

Daniel Gustafsson and Nicolas Pouillard

IT University of Copenhagen, Denmark

Keywords: Agda, Type Theory, Homotopy Type Theory, Formal reasoning, Combinatorics, Isomorphisms, Parametricity, Probabilistic Reasoning

Abstract. Big operators, such as $\sum_{x \in A} f(x)$, $\bigwedge_{x \in A} f(x)$ are the iterated versions of the binary operators $+$ and $\wedge$. They are common in mathematics and we propose tools to reason about them within a Type Theory such as Agda. Using a polymorphic encoding of binary trees one can lift any binary operator to the corresponding big operator. Thanks to the parametricity of this encoding one can easily lift properties of the binary operator to the corresponding big operator. In particular big operators such as sums and products can be put in correspondence with the cardinality of Σ-types and Π-types which enforces a correct implementation. Moreover these correspondences enable the use of type isomorphisms as a powerful reasoning tool. For instance using a standard isomorphism on Σ-types yields a constructive proof that adequate summation functions are invariant under permutations.

1 Introduction

Iteration of binary operators is found in the formalizations of various theories. For any binary operator such as $+$, $\wedge$, max, etc. there is a corresponding iterated version called the big operator ($\sum_{x \in A} f(x)$, $\bigwedge_{x \in A} f(x)$, $\max_{x \in A} f(x)$, etc.) Big operators share a common construction and therefore one should be able to reason about them in a unified fashion. In the Isabelle [1] proof assistant, reasoning about big operators is part of the standard library of theories. However, big operators in Isabelle are built from classical set theory and we wish to adopt a constructive style allowing direct program execution. In the constructive setting, Bertot et al. already studied how to give some properties of big operators given the properties of the small one. Their work led to the `bigops` [2] library using the Coq [3] proof assistant. Given a big operator $\bigoplus_{x \in A} f(x)$, the index set A is a list in the `bigops` library while it is a type in our setting. Since our approach focuses on big operators which iterates over types, it enables a rich use of type isomorphisms. These type isomorphisms are both used to specify and reason about big operators.

By having an abstract theory of big operators we can prove properties which hold for all big operators. It is possible to lift properties from an arbitrary small operator to the corresponding big operator. The following lemma is an example of a property that is easily proven within our framework. Any homomorphism f

acting on two small operators $\oplus$ and $\otimes$ (i.e. $\forall xy,\ f(x \oplus y) \equiv f(x) \otimes f(y)$) can be lifted to the corresponding big operators $\bigoplus$ and $\bigotimes$ working on a body g represented by a function: $f\left(\bigoplus_{x\in A} g(x)\right) \equiv \bigotimes_{x\in A} f(g(x))$.

One use of this lemma is to pick $+$ and $*$ as the small operators, the exponentiation function forms the homomorphism. Another use would be to pick $\wedge$ and $\vee$ on $\{0,1\}$, where the De Morgan law makes $\neg$ the homomorphism:

$$b^{\sum_{a\in A} g(a)} \equiv \prod_{a\in A} b^{g(a)} \qquad\qquad \neg\left(\bigvee_{a\in A} g(a)\right) \equiv \bigwedge_{a\in A} \neg g(a)$$

Uniform discrete probabilities can solely be described by summation functions. Indeed since the universe of events Ω is finite the following holds[1]:

$$\Pr[A|B] \equiv \frac{\Pr[A\cap B]}{\Pr[B]} \equiv \frac{\sharp\,(A\cap B)}{\sharp\,(B)} \equiv \frac{\sum_{x\in\Omega} A(x) \wedge B(x)}{\sum_{x\in\Omega} B(x)}$$

Cardinality in type theory: By *type theory* we mean any dependent type theory similar to Per Martin-Löf type theory [4] or the Calculus of Construction [5]. In such a theory, types have an algebraic structure with $\_\uplus\_$ (disjoint union) and $\_\times\_$ (cartesian product) as addition and multiplication. When looking at the cardinality of types, these corresponds to addition and multiplication. Furthermore Σ-types and Π-types can be seen as big operators for $\_\uplus\_$ and $\_\times\_$, satisfing the following cardinality equations for finite sets:

$$\sharp\,(\Sigma\ \mathtt{A}\ \mathtt{F}) \equiv \sum_{\mathtt{x}\in\mathtt{A}} \sharp\,(\mathtt{F}\ \mathtt{x}) \qquad\qquad \sharp\,(\Pi\ \mathtt{A}\ \mathtt{F}) \equiv \prod_{\mathtt{x}\in\mathtt{A}} \sharp\,(\mathtt{F}\ \mathtt{x})$$

We cannot reason directly with cardinalities inside type theory since its cardinalities are not internalised. However we can still reason about them by using finite types and type isomorphisms, since if two types are isomorphic they have the same cardinality assuming the types are sets. Summation functions can then be related to the cardinality of the corresponding Σ-type. With this correspondence we can show that summations are stable under isomorphisms:

$$\forall (\pi \in A \cong B),\ \sum_{x\in A} f(\pi(x)) \equiv \sum_{y\in B} f(y)$$

We aim at making our results and developments compatible with the univalent foundation of mathematics [6]. More precisely our results should carry over any implementation of homotopy type theory such as Coq [3] and Agda [7]. In particular the `bigops` library could be extended to our use of type isomorphisms. Moreover the univalent setting makes the use of isomorphisms significantly more convenient.

Contributions:

- We give a definition of being an "adequate" summation function, namely that each value is applied exactly once. Adequacy of summation functions

[1] We use $\sharp\, R$ to denote the cardinality of R.

amounts to proving an isomorphism with Σ-types. Furthermore this proof of adequacy, together with standard isomorphisms can be used to simplify proofs as discussed in section 3.

- In section 3.2 we describe how summation functions and isomorphisms can be used to compute uniform discrete probabilities and reason about probabilistic functions.
- Exploration functions (see section 2) implement big operators as higher-order functions. These functions can be combined and transformed to achieve feature-rich explorations in a modular way as discussed in section 4.
- To reason about these big operators we found a new use of the induction principle for binary trees. This principle gives us a way of lifting properties from the small operator to the big operator, such as the lifting of homomorphisms described above. This is discussed in section 4.2.
- For the sake of conciseness, we display only code fragments in the paper. However, a self-contained AGDA development is available online [8].

Notations: Throughout the paper, our definitions are presented in AGDA [7] notation. With `★` we denote the type of types. The function space is written `A → B`, while the dependent function space is written `(x : A) → B x`, `∀ (x : A) → B x`, or `Π A B`. An implicit parameter, can be introduced via `∀{x : A} → B x`, and can be omitted at a call site if its value can be uniquely inferred from the context. There are shortcuts for introducing multiple arguments at once or for omitting a type annotation, as in `∀{A} {i j : A} x → e`. We will use mixfix declarations, such as `_⊎_`, where underscores denote where arguments go. AGDA is strict about whitespace, for instance `explore⊎` is a single identifier because it contains no space.

Core types: As a tool AGDA comes with no predefined concepts other than types and functions, therefore everything has to be defined. In particular there is no specific sort for propositions: everything is in `★`. We denote the empty type as `𝟘` and it is used to represent falsity. The type `𝟙` has one value namely 0_1 and it is used to represent trivial truth. The type `𝟚` has two values (0_2 and 1_2), and it is used both to denote a single bit of information and as a Boolean value where 0_2 denotes false and 1_2 denotes true. We use the type `Fin n` which inductively defines the natural numbers strictly below the natural number `n`. We mainly use this type as a representative for finite types with `n` values. The type `_≡_` is the type of propositional equality also called identity type. AGDA reserves the usual equality symbol `=` for definitions; we apply this convention to our mathematical statements as well.

A note on Σ-types and type isomorphisms: In type theory `Σ A B` is used to denote a dependent sum (sometimes called a dependent pair). Here `A` is a type and `B` is a dependent type over `A` (hence `B` has type `A → ★`). These pairs can be built using the `_,_` constructor (`_,_` has type `(x : A) → B x → Σ A B`). Moreover, pairs come with two projection functions $\texttt{proj}_1$ `: Σ A B → A` and $\texttt{proj}_2$ `: (p : Σ A B) → B (`$\texttt{proj}_1$` p)`. The type `A ≅ B` is used to denote isomorphisms between types `A` and `B`: it means there is two functions, one going from `A → B` and the other from `B → A` such that composing them yields the identity function. The type `_≅_` is an equivalence relation for types.

2 Big operators, folds and explorations

For some of our examples we use the type for six-sided dice, which is introduced by the following Agda declaration. This declaration defines a new data type and thus introduces the type constructor `D6` and its data constructors ⚀, ⚁, ⚂, ⚃, ⚄, and ⚅: In Agda: `data D6 : ★ where ⚀ ⚁ ⚂ ⚃ ⚄ ⚅ : D6`. From the binary operator `_+_` we get the big operator `sumD6` by iterating it over all arguments of the type `D6`. In Agda: `sumD6 f = f ⚀ + (f ⚁ + (f ⚂ + (f ⚃ + (f ⚄ + f ⚅))))`.

All big operators over a type `A` share a common type, namely `(A → U) → U`. For instance summations and products have type `(A → ℕ) → ℕ`, while disjunctions and conjunctions have type `(A → 𝟚) → 𝟚`.

Definition 1. *A big operator for a small operator* `_⊕_` *together with default value* ϵ *of type* `U` *is a function that, given a body of type* `A → U` *for some type* `A`*, will apply some values of* `A` *and combine them with the* `_⊕_` *operator. If there are no values to apply,* ϵ *is returned. In Agda:* `BigOp U A = (A → U) → U`*.*

Definition 2. *An exploration function for a type* `A` *is given a type* `U`*, a value* ϵ *of type* `U`*, a function* `_⊕_` *of type* `U → U → U`*, and function* `f` *of type* `A → U`*. The exploration function finally yields a result of type* `U`*. In Agda:*
`Explore A = ∀ {U : ★} (ϵ : U) (_⊕_ : U → U → U) → BigOp U A`

For *any* type `A`, an exploration function is given a default result ϵ, a binary operator `_⊕_` and a function `f` realising the body of the big operator. The function `f` is then called on every value of the type to be explored. All results are combined with the operator `_⊕_`. If there are no values to explore the default result ϵ is returned. The task of an exploration function is thus to transform any small operator `_⊕_` into the corresponding big operator $\bigoplus$ of type `(A → U) → U`. For instance, if `explore` is an exploration function for a type `A`, then `explore 0 _+_` is $\sum$ and `explore 1 _*_` is $\prod$.

Finiteness: Given Agda's type discipline, the type `Explore A` enforces that any exploration function will only explore a finite number of values (of type `A`). This is enforced by Agda functions being total (strongly normalizing and exhaustively defined) and by parametricity [9, 10]: since the exploration function knows nothing about the type `U` it must use what is given to it.

Exhaustivity: Some exploration functions can be defined to explore all the values of a type `A`. These exploration functions are then said to be *exhaustive*. Originally, the name "exploration" was coined because these functions were designed to systematically examine every possible value of the type.

List folding as an exploration: A straightforward way to build an exploration function is to start with a list. Given a list of values to explore, the corresponding exploration function is not more than a variation of the standard fold on lists. Using a list is a way to explore a known subset of a type: for instance, one can explore the even faces of a die with: `foldList (⚁ :: ⚃ :: ⚅ :: [])`. If one wants the exploration to be exhaustive one has to first produce an exhaustive list of the values of type `A`. However, lists force a flat structure for the exploration:

```
foldList : ∀ {A : ★} → List A → Explore A
foldList []        ϵ _⊕_ f = ϵ
foldList (x :: xs) ϵ _⊕_ f = f x ⊕ foldList xs ϵ _⊕_ f
```

Tree folding are explorations too: One way to gain flexibility is to use trees instead of lists. We define below a classic data type for binary trees which follows the notion of exploration. An exploration function can be seen as the polymorphic Church encoding as studied by Böhm and Berarducci [11]. The function `foldTree : ∀ {A} → Tree A → Explore A` is straightforward and omitted:

```
data Tree (A : ★) : ★ where empty : Tree A
                            leaf  : A → Tree A
                            fork  : (ℓ r : Tree A) → Tree A
```

As an example, given a concrete set of elements represented as a binary tree `t`, the function `foldTree t` is an exploration function for these elements. For instance, `foldTree (node (leaf 3) (node empty (leaf 1))) 0 _+_ (_*_ 2)` computes to the value `(2 * 3) + (0 + (2 * 1))` which finally computes to `8`.

Summing, Counting, and Size: Given an exploration function `explore` on a type `A` we can derive multiple operations from it. In particular, exploration functions give us summation functions by using `0` for ϵ and `_+_` for `_⊕_`. Counting the values satisfing a predicate is a particular case of summation. The function `count` lifts the predicate `p : A → 𝟚` to a summation body of type `A → ℕ`. Additionally, we define `size` which counts using a trival predicate (the constant function 1_2). Provided `explore` is indeed exploring all the values of type `A` exactly once then `size` must be the size of the type `A` (see lemma 1).

Monoidal explorations: When the small operator, together with the default value, is equipped with some structure, the resulting big operator has some interesting properties. For instance, most of the interesting small operations form at least a monoid. This is the case for a many container-like data structures. Among these data structures we highlight three: binary trees, lists, and options (`Maybe`). These enable an exploration to be reified as a data structure. The binary tree structure reifies exactly the exploration, keeping the original balancing. The list type corresponds to the free monoid: it yields a flat representation which is canonical up to monoidal laws. The `Maybe` type can be used to implement the retrieval of the first explored value. Combined with filtering this provides a way to "find" a particular value. These reification functions are the inverses of the folding functions defined above.

3 Cardinality and Type Isomorphism

3.1 Adequate summations through Type Isomorphisms

Our original motivation for exploration functions was to compute probabilities by summing an event over all the values of a given type. However, how can we ensure that we have a correct summation function? We need to ensure that a summation function is going to count every value exactly once (i.e the summation function is not allowed to forget a value or over-count it). In order to guarantee this we use a strong correspondence between the cardinalities of types in type theory and the act of summing. We use this correspondence as a specification for the summation functions that fully explores a type. It boils down to the observation that $\Sigma\ \mathtt{A}\ \mathtt{F}$ is acting as a big operator for disjoint union of all $\mathtt{F}\ \mathtt{x}$ where $\mathtt{x}$ is of type $\mathtt{A}$. Therefore the cardinality of a Σ-type is the summation of the cardinalities over the type family: $\sharp\,(\mathtt{Fin}\ \mathtt{n}) \equiv \mathtt{n}$ and $\sharp\,(\Sigma\ \mathtt{A}\ \mathtt{F}) \equiv \sum_{\mathtt{x}\in\mathtt{A}} \sharp\,(\mathtt{F}\ \mathtt{x})$.

Using these cardinality relations we can show that $\mathtt{sum}^{\mathtt{A}}$ a summation function is correct, assuming a particular type isomorphism exists. Since type isomorphisms preserve cardinality, we argue as follows.

$$\mathtt{sum}^{\mathtt{A}}\ \mathtt{f} \equiv \sharp\,(\ \mathtt{Fin}\ (\mathtt{sum}^{\mathtt{A}}\ \mathtt{f})\ ) \equiv \sharp\,(\ \Sigma\ \mathtt{A}\ (\mathtt{Fin} \circ \mathtt{f})) \equiv \sum_{\mathtt{x}\in\mathtt{A}} \sharp\,(\mathtt{Fin}\ (\mathtt{f}\ \mathtt{x})) \equiv \sum_{\mathtt{x}\in\mathtt{A}} \mathtt{f}\ \mathtt{x}$$

Definition 3. *A function sum^A for a type A is said to be an adequate sum if for all functions f there is an isomorphism between $\Sigma\ A\ (Fin \circ f)$ and $Fin\ (sum^A\ f)$. In* Agda*: $AdequateSum\ sum^A = \forall\ f \to \Sigma\ A\ (Fin \circ f) \cong Fin\ (sum^A\ f)$.*

Using this specification we get a correctness criterion for summation functions and we can use type isomorphisms to derive results about our summation functions. For instance, summation functions are invariant under isomorphism.

Lemma 1. *For an adequate summation function sum over type A, with a derived size $size : \mathbb{N}$, then it is possible to construct an isomorphism $Fin\ size \cong A$.*

Proof. By $\mathtt{sum}$ being adequate and the isomorphism $\Sigma\ \mathtt{A}\ (\lambda\ \_ \to \mathtt{Fin}\ 1) \cong \mathtt{A}$.

Lemma 2. *Let A, B be sets*[2]*, for all $\pi : A \cong B$ and for all C being a type family over B, it is possible to construct an isomorphism $\Sigma\ A\ (C \circ \pi) \cong \Sigma\ B\ C$ (the proof is omitted but can be found in our* Agda *development).*

Theorem 1. *Given two adequate summation functions sum^A and sum^B for types A and B respectively, for all isomorphisms $\pi : A \cong B$ and functions $f : B \to \mathbb{N}$ the summation $sum^A\ (f \circ \pi)$ is equal to the summation $sum^B\ f$.*

[2] Set in the sense of homotopy type theory.

Proof. Using adequacy of the summation functions and the lemmas 1 and 2 one gets an isomorphism $\mathtt{thm} : \mathtt{Fin}\ (\mathtt{sum}^{\mathtt{A}}\ (\mathtt{f} \circ \pi)) \cong \mathtt{Fin}\ (\mathtt{sum}^{\mathtt{B}}\ \mathtt{f})$. Since $\mathtt{Fin}$ is injective (i.e $\mathtt{Fin}\ \mathtt{m} \cong \mathtt{Fin}\ \mathtt{n} \to \mathtt{m} \equiv \mathtt{n}$) the proof is complete.

$$\begin{tikzcd}
\mathtt{Fin}\ (\mathtt{sum}^{\mathtt{A}}\ (\mathtt{f} \circ \pi)) \arrow[r, leftrightarrow, "\text{thm}"] \arrow[d, leftrightarrow, "\text{adequacy of } \mathtt{sum}^{\mathtt{A}}"'] & \mathtt{Fin}\ (\mathtt{sum}^{\mathtt{B}}\ \mathtt{f}) \arrow[d, leftrightarrow, "\text{adequacy of } \mathtt{sum}^{\mathtt{B}}"] \\
\Sigma\ \mathtt{A}\ (\mathtt{Fin} \circ \mathtt{f} \circ \pi) \arrow[r, leftrightarrow, "\text{lemma 2}"'] & \Sigma\ \mathtt{B}\ (\mathtt{Fin} \circ \mathtt{f})
\end{tikzcd}$$

Lemma 3. *Given two summation functions* $\mathtt{sum}^{A}$ *and* $\mathtt{sum}^{B}$ *for type* A *and type* B*, if both are adequate they satisfy the commutation property that* $\mathtt{sum}^{A}\ (\lambda\ a \to \mathtt{sum}^{B}\ (\lambda\ b \to f\ (a\ ,\ b)))$ *is equal to* $\mathtt{sum}^{B}\ (\lambda\ b \to \mathtt{sum}^{A}\ (\lambda\ a \to f\ (a\ ,\ b)))$.

Proof. By adequacy of $\mathtt{sum}^{\mathtt{A}}$ and $\mathtt{sum}^{\mathtt{B}}$ and the following type isomorphism: $(\Sigma\ \mathtt{A}\ \lambda\ \mathtt{x} \to \Sigma\ \mathtt{B}\ \lambda\ \mathtt{y} \to \mathtt{C}\ \mathtt{x}\ \mathtt{y}) \cong (\Sigma\ \mathtt{B}\ \lambda\ \mathtt{y} \to \Sigma\ \mathtt{A}\ \lambda\ \mathtt{x} \to \mathtt{C}\ \mathtt{x}\ \mathtt{y})$.

Counting uniquely: We prove that all values are summed only once when using an adequate summation function $\mathtt{sum}$.

Theorem 2. *Assume for a type* A *that we have a boolean equality test* $\_\mathtt{==}\_$ *such that, for all* x *and* y *of type* A*, the type* $(x == y) \equiv 1_2$ *is isomorphic to* $x \equiv y$*. Furthermore, assume an adequate summation function* $\mathtt{sum}$*, from which we derive a counting function* $\mathtt{count}$*. Then, for all* x*, the equation* $\mathtt{count}\ (\lambda\ y \to x == y) \equiv 1$ *holds.*

Proof. Using the fact that $\mathtt{sum}$ is an adequate summation function together with the type isomorphism $\Sigma\ \mathtt{A}\ (\lambda\ \mathtt{y} \to \mathtt{x} \equiv \mathtt{y}) \cong \mathbb{1}$.

Adequate explorations for free: If an adequate summation function arises from an exploration function, we learn that the exploration function has applied each argument of the type only once. This holds because the type of exploration functions is polymorphic so an exploration function has no way to cheat and detect it supposed to count how many times a certain value occurs. This result holds by parametricity [9, 10] and provides the adequacy of exploration functions.

3.2 Probabilistic functions, deterministically

While a deterministic function is a fixed mapping from elements of a domain A to elements of a codomain B, a probabilistic function carries out a probabilistic process to map the elements of A to the elements of B.

This extra capability of a probabilistic function p can be modeled by a deterministic function f receiving one extra argument r uniformly drawn from a set R. The argument r represents the randomness required by the probabilistic process. When the function f is correctly chosen the following holds for all arguments x and result y: $\Pr[r \leftarrow R; f(x,r) \equiv y] \equiv \Pr[p(x) \equiv y]$.

In this work we focus on a finite random supply R or equivalently a finite universe of events Ω. With this setting one can reason about uniform discrete probabilities using exploration functions and isomorphisms. For a probabilistic function which needs to flip a coin, roll a six-sided die and generate a 128-bits key, the type `R` can be any type isomorphic to $(\mathbb{2} \times \texttt{D6} \times \texttt{Bits}\ 128)$.

Theorem 3. *Assume an adequate summation function* `sum` *over a type* `R` *and let* `count` *be the derived counting function. For two events* $f\ g : R \to \mathbb{2}$, *such that* `f` *and* `g` *have the same probability of occuring i.e* $\texttt{count}\ f \equiv \texttt{count}\ g$, *it is possible to construct an isomorphism* $\pi : R \cong R$ *such that* $f\ x \equiv g\ (\pi\ x)$.

Proof. The isomorphism π is the identity for all values `x` such that $\texttt{f x} \equiv \texttt{g x}$. Otherwise π is defined as the isomorphism between $\Sigma\ \texttt{R}\ (\lambda\ \texttt{x} \to \texttt{f x} \equiv 1_2 \times \texttt{g x} \equiv 0_2)$ and $\Sigma\ \texttt{R}\ (\lambda\ \texttt{x} \to \texttt{f x} \equiv 0_2 \times \texttt{g x} \equiv 1_2)$, which is first derived from the adequacy of `sum` and that $\texttt{count f} \equiv \texttt{count g}$.

Corollary 1. *Two events* $f\ g : R \to \mathbb{2}$ *have the same probability of occurring if and only if there is an isomorphism* $\pi : R \cong R$ *such that* `g` *is equal to* $f \circ \pi$.

Proof. Combining theorems 1 and 3.

Corollary 2. *Uniform distributions: For any type* `A` *and any value* `x` *of type* `A`, *the likelihood of a random sample* `y` *of type* `A` *being equal to* `x` *is* $Pr[x \equiv y] \equiv \frac{1}{\sharp(A)}$.

Proof. Follows directly from theorem 2.

This corollary implies that our definition of random sampling corresponds to a uniform sampling. Uniform distributions are those that attribute the same probability to all values of the type used as the universe of events. For finite types this amounts to saying that each value has to be counted exactly once.

Examples of using isomorphisms for summations: When reasoning about probabilities, one establishes the relation between the probabilities of two processes. A deduction step either approximates (weakens, loosens) this relation or keeps it unchanged. In the latter case the probability stays the same because of a symmetry within the space of events. We exploit these symmetries by revealing isomorphic event spaces.

Examples from cryptography: Internally an encryption scheme often works using group structures. Assuming an arbitrary group $(\texttt{G}, 0, \_\oplus\_, -\_)$, the security of the system often relies on the fact that, for any `x`, adding a random value to `x` will still appear random. The standard example is one time pad where encryption is just bitwise XOR of the key and the message. If one can show that $\lambda\ \texttt{x} \to \texttt{x} \oplus \texttt{m}$ is an isomorphism for some `m` then adding a random value to `m` is indistinguishable from random. This indstinguishability is proven by showing that, for all observers $\texttt{O} : \texttt{G} \to \mathbb{N}$, $\texttt{sum}\ (\lambda\ \texttt{x} \to \texttt{O}(\texttt{x} \oplus \texttt{m}))$ is equal to $\texttt{sum}\ (\lambda\ \texttt{x} \to \texttt{O}(\texttt{x}))$ due to theorem 1. In particular the observer learns nothing of `m`, which is why this provides security.

One case where this reasoning is used is when proving the security of a stream cipher. A stream cipher assumes a pseudo random number generator PRG which is a probabilistic function that will output random looking data. Compared to one time pad, the main benefit of a stream cipher is that the size of randomness required is less than the size of the output. The encryption of a stream cipher is PRG(key) XOR m where m is the message: one usually argues that this is secure because PRG(key) is supposed to be indistinguishable from random.

Another example is in the proof of the ElGamal encryption system which works in a multiplicative group instead. In one part of the proof the adversary gets a ciphertext $c = (g^y , g^z \bullet m)$ where both g^y and g^z can be considered to be random. Hence the adversary will not learn anything about the message m.

3.3 Cardinality of *Π*-types

This correspondence can be further extended to products. *Π*-types can be seen as the big operator for products. The correctness for *product* functions can be defined using correspondence similar to the one for summation functions:

$$\mathtt{prod}^{\mathtt{A}}\ \mathtt{f} \equiv \sharp\left(\mathtt{Fin}(\mathtt{prod}^{\mathtt{A}}\ \mathtt{f})\right) \equiv \sharp\left(\Pi\ \mathtt{A}(\mathtt{Fin} \circ \mathtt{f})\right) \equiv \prod_{\mathtt{x} \in \mathtt{A}} \sharp\left(\mathtt{Fin}(\mathtt{f}\ \mathtt{x})\right) \equiv \prod_{\mathtt{x} \in \mathtt{A}} \mathtt{f}\ \mathtt{x}$$

Definition 4. *A function $prod^A$ for a type A is said to be an adequate product if for all f there is an isomorphism between Π A (Fin ∘ f) and Fin ($prod^A$ f). In* Agda*: AdequateProduct $prod^A$ = ∀ f → Π A (Fin ∘ f) ≅ Fin ($prod^A$ f).*

However, this definition asks for an isomorphism with a function space, and these can sometimes be difficult to establish without functional extensionality. Such difficulties are evident in simple examples, like trying to construct the isomorphism $\Pi\ \mathbb{2}\ \mathtt{B} \cong \mathtt{B}\ 0_2 \times \mathtt{B}\ 1_2$ where B is a dependent family over $\mathbb{2}$.

One promising solution to the problem of functional extensionality in a constructive setting is homotopy type theory [6] which has generated much interest in recent years. This theory includes the univalence axiom, which states that homotopy equivalence of types is homotopically equivalent to identity of types: as a consequence we get that equality of functions is extensional equality. We will for the rest of this section assume we are working in a setting with functional extensionality like homotopy type theory.

Theorem 4. *Let $prod^A$ be an adequate product function for the type A. Let sum^{AB} be an adequate summation function for a type Π A B. Finally let sum^B be a family over A of summation functions on the type B. Then for all function f : (x : A) → B x → ℕ, $prod^A$ (λ x → sum^B x (λ y → f x y)) is equal to sum^{AB} (λ g → $prod^A$ (λ x → f x (g x))).*

Proof. Using the adequacy properties together with the type isomorphism
Π A (λ x → Σ (B x) λ y → C x y) ≅ Σ (Π A B) λ f → Π A λ x → C x (f x)
(the forward direction is usually known as the dependent axiom of choice).

4 Working with exploration functions

Exploration functions can be obtained by folding over data structures such as lists or trees. However, one can also define exploration functions directly. This corresponds to the polymorphic encoding for binary trees. In this section we show how to build, combine, transform, and reason directly about these. Below `exploreD6` is an example of an exploration function for `D6` defined directly.

```
exploreD6 : Explore D6
exploreD6 ϵ _⊕_ f = f ⚀ ⊕ (f ⚁ ⊕ (f ⚂ ⊕ (f ⚃ ⊕ (f ⚄ ⊕ f ⚅))))
```

Building exploration functions: In order to easily define new exploration functions we provide three building blocks inspired by binary trees. These three combinators are defined for any type `A` and correspond to the constructors `empty`, `leaf`, and `fork`. The function `empty-explore` is an exploration function which does not explore anything and just returns the default value ϵ. The function `point-explore` takes a value `x` of type `A` and defines an exploration function which explores only this point `x` using the given exploration body. Finally the function `merge-explore` takes two exploration functions and combines them using the received binary operator `_⊕_`.

For exhaustively exploring finite types, however, we have more specialised combinators. Generally, finite types are a combination of sums and products, therefore exploration combinators are provided for those. As base cases we have exploration functions for types such as $\mathbb{0}$, $\mathbb{1}$ and $\mathbb{2}$. For sum types `A ⊎ B`, the exploration `explore⊎` $\texttt{exp}^{\texttt{A}}$ $\texttt{exp}^{\texttt{B}}$ ϵ `_⊕_ f` combines the two results given by exploring the function `f` specialised to types `A` and `B` using $\texttt{inj}_1$ and $\texttt{inj}_2$ — the injections for the type `_⊎_`. The two results are then combined using `_⊕_`. For Cartesian products `A × B`, the exploration `explore×` $\texttt{exp}^{\texttt{A}}$ $\texttt{exp}^{\texttt{B}}$ ϵ `_⊕_ f` nests the exploration of `B` into the function exploring `A`. Note how this combinator is independent of the operator `_⊕_`. Finally `explore×` is generalised to Σ-types (dependent sums) by `explore`Σ. Intuitively, we take the following equation on summations as our definition for exploration of product types:

$$\sum_{x,y \in A \times B} f(x,y) \equiv \sum_{x \in A} \sum_{y \in B} f(x,y)$$

Exploration functions for exploring functions: While we found no way to directly explore functions themselves (such as `Explore (A → B)`) there is an attractive workaround: one can use isomorphisms on functions to incrementally build such an exploration function. Mainly one decomposes the domain with isomorphisms towards simpler types we can explore:

$$(\mathtt{A} \uplus \mathtt{B}) \to \mathtt{C} \cong (\mathtt{A} \to \mathtt{C}) \times (\mathtt{B} \to \mathtt{C}) \qquad (\mathtt{A} \times \mathtt{B}) \to \mathtt{C} \cong \mathtt{A} \to (\mathtt{B} \to \mathtt{C})$$

These isomorphisms require functional extensionality, making this one more case where homotopy type theory can help. While not required to define the exploration functions themselves, the proofs of these isomorphisms are required to prove their adequacy.

4.1 Exploration transformers

In this section we describe a series of transformations on exploration functions. These tools provide ways to enhance explorations in a modular way.

We start with an example which highlights how exploration functions may be used as a programming tool in rather than a reasoning tool. A prototypical program involving an exploration function in the cryptography setting is the brute force exhaustive search. This could be the search for a key such that the message successfully decrypts to a meaningful message. Sometimes the message space is relatively small and searching it can be used to gather information, for example inverting hashing functions. Here let us suppose a type for messages `Msg` together with an exploration function `explore`^{M}, a type for digests `Digest` together with an equality test (`_==_` has type `Digest → Digest → 𝟚`), and a function `H : Msg → Digest`. In practice one might think of the function `H` as being hard to inverse. The following program inverts `H` by exploring all possible messages, and returning the list of all messages which maps to the input digest:

```
H⁻¹-list : Digest → List Msg
H⁻¹-list d = exploreᴹ [] _++_ λ m → if H m == d then [ m ] else []
```

In order to prove probabilistic results about an attacker like `H⁻¹-list` the reasoning might involve the relation between two nested explorations. The outer exploration counts the probabilities and the inner one is a brute force search over all messages. The properties which underlie our exploration functions provide a solid basis for this form of reasoning. While straightforward, the exploration in `H⁻¹-list` shows a lack of modularity: indeed the choice of producing a list of the messages is entangled with the filtering.

Explorations can be chained in such a way that each explored value of type `A` can yield a nested exploration on a type `B`. The resulting exploration aggregates all the spawned explorations and yields results of type `B`:

```
_>>=_ : Explore A → (A → Explore B) → Explore B
(expᴬ >>= expᴮ) ε _•_ f = expᴬ ε _•_ λ x → expᴮ x ε _•_ f
```

Explorations are monadic: The suggestive name (`_>>=_`) highlights that `Explore` forms a monad, where `point-explore` plays the unit (or return). This monadic structure comes as no surprise once we realise that the type `BigOp U` is the continuation monad.

The function `gfilter-explore` (for generic filter) discards undesirable values and also selects what parts to retain from the desirable ones. Using `_>>=_`

filtering is nicely expressed, by chaining the exploration on the type `A` with either `empty-explore` or `point-explore` depending on the explored value `x`. By lifting the given function `f` to a predicate the function `filter-explore` uses `gfilter-explore`.

```
gfilter-explore : (f : A → Maybe B) → Explore A → Explore B
filter-explore  : (p : A → 𝟚) → Explore A → Explore A
```

The previous example inverting a function `H` can built using `filter-explore (λ m → H m == d)` the result is an exploration from which on can get a list, or the first matching values.

A rather trivial exploration transformer is `explore-backward`, which flips the arguments of the given small operator. With this function we emphasis how monoid transformers (such as `flip`) yield exploration transformers.

```
explore-backward : Explore A → Explore A
explore-backward eᴬ ϵ _●_ = eᴬ ϵ (flip _●_)
```

As a last transformer we consider the monoid of endomorphisms featuring the identity function as the neutral element and function composition as the multiplication operation. Exploring with the monoid of endomorphisms expects a function body that will turn values of type `A` into functions of type `U → U`. The body composes the original small operator `op` with the original body `f`. We finally pass in the default value ϵ to the resulting big composition. When `(ϵ , _●_)` is a monoid, this transformation computes to the same result as the original exploration. Its utility lies in the fact that function composition has an associative computational content which will force all the calls to `_●_` to be associated to the right, finally ending with a single ϵ. This technique, known as *difference lists*, (has been used before and) is part of the standard toolbox of functional programmers. Its original motivation was to improve the performance, but it is also useful for reasoning since it gives associativity for free. Notice that this technique is nicely captured by the following exploration transformer:

```
explore-endo : Explore A → Explore A
explore-endo eᴬ ϵ op f = eᴬ id _∘_ (op ∘ f) ϵ
```

4.2 Exploration Principle

We build the reasoning principle of exploration functions on top of binary tree induction, so we first review binary tree induction. As with any induction principle, the desired property `P`, must be proved for the base cases, namely `empty` and `leaf x` here. The property `P` must then be preserved by the recursive structure, namely `node` here.

Definition 5. *The exploration principle states that any property* `P` *on exploration functions holds if* `P` *holds for* `empty-explore`*; if* `P` *holds for all points (using* `point-explore`*); and if* `P` *is preserved by* `merge-explore`*. In* Agda*:*

```
ExplorePrinciple : ∀ {A} → Explore A → ★
ExplorePrinciple {A} exp =
  ∀ (P   : Explore A → ★) (εᴾ : P empty-explore)
    (⊕ᴾ  : ∀ {e₀ e₁} → P e₀ → P e₁ → P (merge-explore e₀ e₁))
    (fᴾ  : ∀ x → P (point-explore x)) → P exp
```

Proper exploration functions comes with the principle defined above. This principle is the induction principle on binary trees where `empty`, `node`, and `leaf`, respectively become `empty-explore`, `merge-explore` and `point-explore`. Put differently, this property enforces that an exploration function is essentially a binary tree where empty trees are ϵ, nodes are calls to `_⊕_`, and leaves are calls to `f`. This principle alone implies most of the properties which are expected from big operators (apart from the unique counting property of theorem 2).

Moreover, while the type of the principle (i.e. `ExplorePrinciple`) looks a bit daunting, it is a simple mechanical process to prove it: one mimics what happens in the underlying exploration function. Below is the actual Agda proof term of this principle for our `exploreD6` function. Thanks to implicit parameters the proof term `exploreD6ᴾ` is almost like `exploreD6`:

```
exploreD6ᴾ : ExplorePrinciple exploreD6
exploreD6ᴾ P εᴾ _⊕ᴾ_ fᴾ
  = fᴾ ⚀ ⊕ᴾ (fᴾ ⚁ ⊕ᴾ (fᴾ ⚂ ⊕ᴾ (fᴾ ⚃ ⊕ᴾ (fᴾ ⚄ ⊕ᴾ fᴾ ⚅))))
```

We conjecture the principle to be provable for each well typed exploration function, following a parametricity result. However this seems to require a slightly stronger variant of free theorems [9] than those proposed by Bernardy et al. [10].

Properties of exploration functions: We have proved some properties inspired by measure theory. These properties are immediate consequences from the exploration principle and properties of the small operator. Examples of such properties are: the exploration is homomorphic provided that the operation is associative and commutative; linearity requires that the operation `_·_` distributes over `_⊕_`; extensionality holds unconditionally; and monotonicity requires that the operation is monotonic with respect to a preorder `_≤_`.

As highlighted in the introduction, a more advanced property connects the exploration of two different operations. The property `lift-hom` lifts an homomorphism to its big operator counterpart. Let `_⊕_` and `_⊗_` be respectively `_+_` and `_*_` on ℕ, `f` be an exponential function $x \mapsto b^x$ for some base b, then the following equation holds:

$$\forall g \in (A \to \mathbb{N}),\, b^{\sum_{x\in A} g(x)} \equiv \prod_{x\in A} b^{g(x)}$$

Given these same properties one can also derive the rule of addition for probabilities. The property proved in AGDA follows from this equation:

$$\sharp(P) + \sharp(Q) \equiv \sharp(P \cup Q) + \sharp(P \cap Q)$$

5 Discussion

5.1 Related work

The Big_Operators theory in Isabelle: Another development of big operators can be found in Isabelle [12]. This library uses an axiomatization of finite sets and a fold function operating on these sets. Since Isabelle/HOL is based on classical logic, the fold function are, in contrast to our exploration functions, not constructive. Because of this we can't directly use the results from this library.

Canonical big operators: The work on the `bigops` library [2] for COQ have a similar purpose as our exploration functions. This library focuses on the properties one can derive about folds over lists. These folds also allow one to filter out undesired values:

```
reduceBig : ∀ {U A : ★}(_⊕_ : U → U → U)(ϵ : U)(ℓ : List A)
           (p : A → 𝟚)(f : A → U) → U
reduceBig _⊕_ ϵ ℓ p f =
  foldr (λ i x → if p i then f i ⊕ x else x) ϵ ℓ
```

By rearranging the types to put the predicate and the list as the first argument we can see that this is indeed a way to construct an exploration function, (although we abstract out the filtering using `filter-explore` from section 4.1). Another way of defining `reduceBig` would be `reduceBig p ℓ = filter-explore p (foldList ℓ)`.

But `foldList` is not the only way of constructing exploration functions: we have a choice and, as such, can pick one that has better reduction behaviour. Take as a motivating example the list corresponding to `explore⊎`. This construction depends on two list functions, `map` and `_++_`, both of which might hinder reduction.

```
combine-⊎ : ∀ {A B : ★} → List A → List B → List (A ⊎ B)
combine-⊎ xs ys = map inj₁ xs ++ map inj₂ ys
```

The issue regarding the reduction is that `reduceBig _⊕_ ϵ (xs ++ ys) p f` will not be definitionally equal to `reduceBig ...xs... ⊕ reduceBig ...ys...` when `xs` is a neutral term. A similar problem occurs when using a `map` as in `reduceBig _⊕_ ϵ (map g xs) p f` which will not be definitionally equal to `reduceBig _⊕_ ϵ xs (p ∘ g) (f ∘ g)`. When defining the exploration functions directly, this reduction is the definition so one are saved from having to work with propositional equality proofs.

In bigops [2], the type finType is characterised by a list together with a proof that, for all element x of that list, x occurs only once i.e. count (_==_ x) xs ≡ 1. Theorem 2 states that every adequate exploration satisfies this criterion.

The Alea *library:* The Coq library Alea [13] is used to reason about probabilities. Instead of summations they extract measures from a monad called Distr. The measure is extracted with the function μ : Distr A → (A → [0,1]) $\rightarrow^{\mathtt{m}}$ [0,1]. Here [0,1] represents the real numbers between 0.0 and 1.0, and $\_\rightarrow^{\mathtt{m}}\_$ represents monotonic functions. For a μ function to be a probability distribution it needs to be a linear continuous operation. The type [0,1] had to be partly axiomatised and as such is not fully computable.

Since we can also sample over finite types in Alea, we can embed probabilistic functions from our system to the Alea monad (Distr). To do so we use the underlying deterministic function. For instance, consider f : R → 𝟚. Once embedded in Alea, we conjecture that the following relation between the probability distribution and our summation functions $\mathtt{sum}^{\mathtt{R}}$ f holds[3]:

```
embed : (R → 𝟚) → Distr 𝟚
embed f = do r ← randᴿ; return (f r)
embedding : ∀ f → μ (embed f) 𝟚▷[0,1] ≡ sumᴿ f / #R
```

5.2 Future work

Big operators over types: Intuitively Σ is the big operator for _⊎_, can we, for a type A, find an exploration function $\mathtt{exp}^{\mathtt{A}}$ such that $\mathtt{exp}^{\mathtt{A}}$ 𝟘 _⊎_ F ≅ Σ A F? Such an exploration function would lay down all the "paths" to the values in the type A. Each of these paths lead to an associated leaf F which depends on the corresponding A value. We conjucture that it is enough to be an adequate exploration function for this to be true.

Higher inductive types: When looking at big operators we usually do not consider the order the elements are applied in to be of importance. This is reflected in the set-theoretical syntax $\bigoplus_{x \in A} f(x)$ that we have used so far. However, nothing prevents us from folding over a non-commutative and non-assocative operator. The tree type described in section 2 allows us to distinguish based on the order of elements. To remedy this one can instead use a higher inductive type [14]. The inductive type of binary trees (Tree) can be upgraded to a higher inductive type (CTree) where the laws for commutative monoids are added as extra equalities. This type CTree corresponds to the free commutative monoid. We conjecture that the induction on the type CTree corresponds to a refienment of exploration functions where the operator enjoys a commutative monoid structure.

[3] We silently coerce $\rightarrow^{\mathtt{m}}$ to → and 𝟚▷[0,1] is measuring the likelihood of getting 1_2

5.3 Conclusion

This work presents a way to reason formally about big operators. We define exploration functions using a polymorphic encoding of binary trees, which offer greater modularity. Reusing the induction principle of binary trees, we are able to easily lift properties from small operators to big operators. Some exploration functions can be shown to adequate, namely each value is applied exactly once, thanks to the cardinality of Σ-types being a sum of cardinalities. We show how to model probabilistic functions directly into type theory. For finite types we use a computational meaning for the probabilities that simplifies the proofs since probabilistic equivalences can be given as isomorphisms between the corresponding Σ-types without involving numbers. Furthermore we made a persistent usage of type isomorphisms to make our results simpler and more general.

References

1. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/hol: A proof assistant for higher-order logic (2002)
2. Bertot, Y., Gonthier, G., Ould Biha, S., Pasca, I.: Canonical big operators. In Mohamed, O., Muñoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics. Volume 5170 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 86–101
3. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004) Version 8.0.
4. Martin-Löf, P.: An intuitionistic theory of types: Predicative part. In Rose, H., Shepherdson, J., eds.: Proceedings of the Logic Colloquium '73. Volume 80 of Studies in Logic and the Foundations of Mathematics. (1975)
5. Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. **76**(2-3) (February 1988) 95–120
6. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. (August 2013)
7. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
8. Gustafsson, D., Pouillard, N.: crypto-agda (2012-2014) https://github.com/crypto-agda/crypto-agda.
9. Wadler, P.: Theorems for free! In: Conference on Functional Programming Languages and Computer Architecture (FPCA). (September 1989) 347–359
10. Bernardy, J.P., Jansson, P., Paterson, R.: Parametricity and dependent types. In: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming. ICFP '10, New York, NY, USA, ACM (2010) 345–356
11. Böhm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. Theor. Comput. Sci. **39** (1985) 135–154
12. Nipkow, T., Paulson, L.C., Wenzel, M.: Theory big_operators http://isabelle.in.tum.de/library/HOL/Big_Operators.html.
13. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in coq. Science of Computer Programming **74**(8) (2009) 568 – 589
14. Lumsdaine, P., Shulman, M.: Higher inductive types. (2013) In preparation.

A Agda development, selected parts

A.1 Fold over binary trees

```
{- Fold over binary trees: -}
foldTree : ∀ {A} → Tree A → Explore A
foldTree empty      = empty-explore
foldTree (leaf x)   = point-explore x
foldTree (fork ℓ r) = merge-explore (foldTree ℓ) (foldTree r)
```

A.2 Reifications

```
tree : Tree A
tree = exploreᴬ empty fork leaf

list : List A
list = exploreᴬ List.[] List._++_ List.[_]

first : Maybe A
first = exploreᴬ nothing _||?_ just
  where
    _||?_ : Maybe A → Maybe A → Maybe A
    nothing  ||? my = my
    (just x) ||? _  = just x
```

A.3 Derived big operators

```
module Explorable {A}(exploreᴬ : Explore A) where
  sum : (A → ℕ) → ℕ
  sum = exploreᴬ 0 _+_

  count : (A → 𝟚) → ℕ
  count f = sum (𝟚▹ℕ ∘ f) -- 𝟚▹ℕ converts 𝟚 into ℕ

  size : ℕ
  size = count (const 1₂)

  product : (A → ℕ) → ℕ
  product = exploreᴬ 1 _*_
```

A.4 Listing of Type Isomorphisms

`((_×_ , 𝟙) , (_⊎_ , 𝟘))` is a commutative semiring up to isomorphism.

```
Fin-inj : Fin m ≅ Fin n → m ≡ n
```

```
Fin-0-𝟘  : Fin 0 ≅ 𝟘
Fin-1-𝟙  : 1 ≅ 𝟙
Fin-2-𝟚  : 2 ≅ 𝟚
Fin-+-⊎  : Fin (m + n) ≅ Fin m ⊎ Fin n
Fin-*-×  : Fin (m * n) ≅ Fin m × Fin n
Fin-Σ    : Fin (sumᴬ f) ≅ Σ A (Fin ∘ f)
Fin-Π    : Fin (prodᴬ f) ≅ Π A (Fin ∘ f)
```

```
Σ-𝟙    : Σ 𝟙 F ≅ F 0₁
Σ-𝟚    : Σ 𝟚 F ≅ F 0₂ ⊎ F 1₂
Σ-⊎    : Σ (A ⊎ B) F ≅ Σ A (F ∘ inj₁) ⊎ Σ B (F ∘ inj₂)
Σ-Σ    : Σ (Σ A B) F ≅ Σ A (λ a → Σ (B a) (λ b → F (a , b)))
Σ-≡    : (x : A) → Σ A (_≡_ x) ≅ 𝟙
Σ-fst : (π : A ≅ B) → Σ A (F ∘ π) ≅ Σ B F
Σ-snd : ((x : A) → B x ≅ C x) → Σ A B ≅ Σ A C
Σ-swp : Σ A λ x → Σ B λ y → C x y ≅ Σ B λ y → Σ A λ x → C x y
```

```
Π-𝟘    : Π 𝟘 A ≅ 𝟙
Π-𝟙    : Π 𝟙 A ≅ A 0₁
Π-𝟚    : Π 𝟚 A ≅ A 0₂ × A 1₂
Π-⊎    : Π (A ⊎ B) C ≅ Π A (C ∘ inj₁) × Π B (C ∘ inj₂)
Π-Σ    : Π (Σ A B) C ≅ (x : A) (y : B x) → C (x , y)
Π-swp : Π A λ x → Π B λ y → C x y ≅ Π B λ y → Π A λ x → C x y
```

```
dep-AC : (x : A) → Σ (B x) λ y → C x y
       ≅ Σ (Π A B) λ f → (x : A) → C x (f x)
```

A.5 Exploration functions

```
BigOp : ★ → ★ → ★
BigOp U A = (A → U) → U

Explore : ★ → ★
Explore A = ∀ {U : ★} (ε : U) (_⊕_ : U → U → U) → BigOp U A

empty-explore : ∀ {A} → Explore A
empty-explore x ε _⊕_ f = ε

point-explore : ∀ {A} → A → Explore A
point-explore x ε _⊕_ f = f x

merge-explore : ∀ {A} → Explore A → Explore A → Explore A
merge-explore e₀ e₁ ε _⊕_ f = (e₀ ε _⊕_ f) ⊕ (e₁ ε _⊕_ f)

explore⊎ : ∀ {A B} → Explore A → Explore B → Explore (A ⊎ B)
explore⊎ eᴬ eᴮ ε _⊕_ f = (eᴬ ε _⊕_ (f ∘ inj₁)) ⊕ (eᴮ ε _⊕_ (f ∘ inj₂))

explore× : ∀ {A B} → Explore A → Explore B → Explore (A × B)
explore× eᴬ eᴮ ε _⊕_ f = eᴬ ε _⊕_ (λ a → eᴮ ε _⊕_ (λ b → f (a , b)))

exploreΣ : ∀ {A} {B : A → ★} → Explore A → (∀ {a} → Explore (B a))
         → Explore (Σ A B)
exploreΣ eᴬ eᴮ ε _⊕_ f = eᴬ ε _⊕_ (λ a → eᴮ {a} ε _⊕_ (λ b → f (a , b)))
```

A.6 Some properties derived from the exploration principle

```
module DerivedProperties {A} (exp : Explore A) (ep : ExplorePrinciple exp)
                         ε _⊕_ ι _⊗_ _·_ _≤_ f g where
  big-⊕ = exp ε _⊕_
  big-⊗ = exp ι _⊗_

  homomorphic  : CommutativeMonoid ε _⊕_ →
                 big-⊕ (λ x → f x ⊕ g x) ≡ big-⊕ f ⊕ big-⊕ g
  extensional  : (∀ x → f x ≡ g x) → big-⊕ f ≡ big-⊕ g
  monotonic    : Monotonic _⊕_ _≤_ → (∀ x → f x ≤ g x) →
                 ε ≤ ε → big-⊕ f ≤ big-⊕ g
  linear       : (∀ k → k · ε ≡ ε) →
                 (∀ k x y → k · (x ⊕ y) ≡ k · x ⊕ k · y) →
                 ∀ k → big-⊕ (λ x → k · f x) ≡ k · big-⊕ f
  lift-hom     : (f ι ≡ ε) → (∀ x y → f (x ⊕ y) ≡ f x ⊗ f y) →
                 f (big-⊕ g) ≡ big-⊗ (f ∘ g)
```

A.7 Exploration transformers

```
gfilter-explore : (A → Maybe B) → Explore A → Explore B
gfilter-explore f eᴬ = eᴬ >>= λ x → case (f x) of λ
                        { nothing  → empty-explore
                        ; (just y) → point-explore y }

filter-explore : (A → 𝟚) → Explore A → Explore A
filter-explore = gfilter-explore λ x → [0: nothing
                                        1: just x ] (p x)
```

A.8 Binary tree induction

```
{- Fold over binary trees: -}
foldTree : ∀ {A} → Tree A → Explore A
foldTree empty      = empty-explore
foldTree (leaf x)   = point-explore x
foldTree (fork ℓ r) = merge-explore (foldTree ℓ) (foldTree r)

{- Induction over binary trees: -}
binTree-ind : ∀ {A} (t  : Tree A)
                    (P  : Tree A → ★)
                    (ϵᴾ : P empty)
                    (⊕ᴾ : ∀ {t₀ t₁} → P t₀ → P t₁ → P (node t₀ t₁))
                    (fᴾ : ∀ x → P (leaf x)) → P t
binTree-ind empty      P ϵᴾ ⊕ᴾ fᴾ = ϵᴾ
binTree-ind (leaf x)   P ϵᴾ ⊕ᴾ fᴾ = fᴾ x
binTree-ind (fork ℓ r) P ϵᴾ ⊕ᴾ fᴾ = ⊕ᴾ (binTree-ind ℓ P ϵᴾ ⊕ᴾ fᴾ)
                                       (binTree-ind r P ϵᴾ ⊕ᴾ fᴾ)
```

A.9 Higher inductive type for binary trees with commutative monoidal structure, in a fictional version of Agda:

```
data CTree (A : ★) : ★ where
  empty   : CTree A
  leaf    : A → CTree A
  node    : (ℓ r : CTree A) → CTree A
  comm    : ∀ x y   → node x y          ≡ node y x
  assoc   : ∀ x y z → node (node x y) z ≡ node x (node y z)
  neutral : ∀ x     → node empty x      ≡ x
```