Foldable containers and dependent types

Daniel Gustafsson
IT University of Copenhagen
dagu@itu.dk

Nicolas Pouillard
IT University of Copenhagen
npou@itu.dk

Abstract

Functional programs using foldable containers need reasoning tools as they are not equipped with laws. Moreover we want to allow any finite type to be foldable as well.

Folding over all the values of a finite type is particularly interesting in a dependent type theory which features Π and Σ types.

Our solution uses parametricity to show how foldable containers relate to monoid homomorphisms. Our development is implemented and verified within the type theory of Agda which is compatible with parametricity.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

Keywords Agda, Foldable Containers, Formal reasoning, Functional Programming, Homotopy Type Theory, Parametricity, Type Isomorphisms, Type Theory

1. Introduction

Folds (or catamorphisms) are a fundamental part of the structure of functional programs. Intuitively, they provide a way to summarize or *reduce* a data container down to a single value.

A large body of the different ways to reduce a data container is captured by the notion of monoid. Monoids arise quite naturally and are ubiquitous in programming, especially in functional programming. `Monoid` is one of the standard type classes in HASKELL. We recall here its definition[1]:

```
class Monoid m where
  ϵ   :: m
  (⊕) :: m → m → m
  {- identity:      ϵ ⊕ x == x ⊕ ϵ == x -}
  {- associativity: x ⊕ (y ⊕ z) == (x ⊕ y) ⊕ z -}

  mconcat :: [m] → m
  mconcat = foldr (⊕) ϵ
```

The function `mconcat` then takes any list and reduce it down to a single value using the monoid operations. While lists are extensively used in functional programming they are hardly the only data container available. Reducing another data container can be achieved by first producing a list and then reducing it. However one might wish to directly reduce the container and thus provide a specialised `mconcat`/`foldr` function. This generalisation has been made in the `Foldable` type class[2], which is displayed below:

```
class Foldable t where
  foldMap :: Monoid m ⇒ (a → m) → t a → m
  foldr   :: (a → b → b) → b → t a → b
```

The minimal complete definition is to define one of these functions, as they can be implemented in terms of each other. The `foldMap` function almost has the same type as `mconcat`. The difference is that the elements in the container do not have to form a monoid, only that it is possible to map them to a monoid.

In contrast with the `Monoid` class, there are no laws associated with the `Foldable` class. This might be discouraging but as we will see it is still possible to reason about programs that uses the `Foldable` class. For example, one of potential laws for `foldr` is that its application should have the same effect as producing a list and applying `List.foldr` on it. But it can be proven that any type correct term will satisfy this law by parametricity. Further examples highlighting the use of parametricity to prove similar results will be shown using the language AGDA [12].

One of the results following from parametricity is how a monoid homomorphism distributes over a fold. Here, a monoid homomorphism is represented by a `newtype MonoidHom` which wraps a function between two monoids respecting the monoid structure. The property is here presented as a property in the style of QuickCheck [4].

```
newtype MonoidHom m n =
  MonoidHom { hom :: m → n }
  {- hom ϵ = ϵ -}
  {- hom (x ⊕ y) = hom x ⊕ hom y -}

distHomProp :: (Monoid m, Monoid n, Foldable t, Eq n)
  ⇒ MonoidHom m n → (a → m) → t a → Bool
distHomProp (MonoidHom h) f t
  = h (foldMap f t) == foldMap (h ∘ f) t
```

A mathematical example of this property is to pick the exponentiation function as the monoid homomorphism from ($\mathbb{N}$,0,+) to ($\mathbb{N}$,1,*). Another example is to pick boolean negation ($\neg$) as the monoid homomorphism from ({0,1},1, $\wedge$) to ({0,1},0, $\vee$) which follows by the De Morgan law:

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

[1] We use ϵ and $(\oplus)$ instead of `mempty` and `mappend`

[2] The actual `Foldable` type class has more methods with default implementations which we elide for concision here.

TODO, TODO.

ACM TODO. . . $15.00.
http://dx.doi.org/10.1145/TODO

Contributions:

- We describe exploration functions, a class of folds which arise from the use of `foldMap`, we show how they can be combined and transformed to achieve feature-rich explorations in a modular way as discussed in Section 2.
- We make a precise account (Section 3) of the parametricity results (a.k.a. free-theorems) of these exploration functions. These results automatically apply to any well typed instance of the type class `Foldable` because of the polymorphic type of `foldMap`. We show how monoid homomorphisms distribute over explorations among other algebraic properties.
- We show how dependent types enable to explore not only values, but also types (Section 4). In particular we show how some explorations exactly correspond to Π and Σ types.
- We describe a class of summation functions (a particular instance of fold) which we call adequate. These adequate summation functions can be used to compute uniform discrete probabilities and reason about probabilistic functions (Section 5.2). In particular we show how adequacy together with type equivalences can lead to elegant proofs (Section 5).
- For the sake of readability, we display only code fragments in the paper. However, a self-contained AGDA development is available online [5, 6]. Moreover, the results (those not involving dependent types), should hold in HASKELL using `foldMap`[3].

Notations: In the remaining part of the paper, our definitions are presented in AGDA [12] notation. With $\star$ we denote the type of types. The function space is presented as $A \to B$, while the dependent function space is presented as $(x : A) \to B\ x$, $\forall\ (x : A) \to B\ x$, or $\Pi\ A\ B$. An implicit parameter, can be introduced via $\forall\{x : A\} \to B\ x$, 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 $\forall\{A\}\ \{i\ j : A\}\ x \to e$. We will use mixfix declarations, such as $\_\uplus\_$, 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 $\star$. The empty type is denoted as $\mathbb{0}$ and used to represent falsity. The type family $\neg : \star \to \star$ is the logical negation, $\neg\ A$ is defined as $A \to \mathbb{0}$. The type $\mathbb{1}$ has one value namely 0_1 and it is used to represent trivial truth. The type $\mathbb{2}$ 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. The type family $\checkmark : \mathbb{2} \to \star$ maps 0_2 to $\mathbb{0}$ and 1_2 to $\mathbb{1}$. We use the type `Fin n` which inductively defines the natural numbers strictly below `n`. We mainly use this type as a representative for finite types with `n` values. The type $\_\uplus\_ : \star \to \star \to \star$ corresponds to the HASKELL type `Either`, the constructors are `inl` and `inr`. We use $\_\uplus\_$ both as a constructive disjunction and as a disjoint union. The type family $\mathtt{Dec} : \star \to \star$ is the type of decidable types, `Dec A` is equivalent to $A \uplus \neg\ A$. The type family $\_\equiv\_$ is the type of propositional equality, also called the identity type. AGDA reserves the usual equality symbol $=$ for definitions; we apply this convention to our mathematical statements as well.

[3] We assume the standard hypothesis about type class laws and restricting to safe features.

A note on Σ-types and type equivalences: In type theory $\Sigma\ A\ B$ 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 $A \to \star$). These pairs can be built using the $\_,\_$ constructor ($\_,\_$ has type $(x : A) \to B\ x \to \Sigma\ A\ B$). Moreover, pairs come with two projection functions $\mathtt{fst} : \Sigma\ A\ B \to A$ and $\mathtt{snd} : (p : \Sigma\ A\ B) \to B\ (\mathtt{fst}\ p)$. The type $A \simeq B$ is used to denote equivalences between types `A` and `B`. To be precise we use the half-adjoint equivalences. An equivalence is therefore made of two functions, $f : A \to B$ and $g : B \to A$, two homotopies $\epsilon : \forall\ x \to f\ (g\ x) \equiv x$ and $\eta : \forall\ x \to g\ (f\ x) \equiv x$, and final homotopy for coherence: $\tau : \forall\ x \to f\ (\eta\ x) \equiv \epsilon\ (f\ x)$. The type $\_\simeq\_$ is an equivalence relation for types.

Remarks on function extensionality and univalence: A Type Theory is said to support function extensionality when functions equal at every point are considered equal. Namely when the following statement is provable: $\forall\ f\ g \to (\forall\ x \to f\ x \equiv g\ x) \to f \equiv g$. Pure Intensional Type Theory does not have a proof of function extensionality, even in the case where both domain and codomain are finite. Indeed in pure Intensional Type Theory a close proof of the identity type must be the reflexivity witness, hence only functions definitionally equal can be shown to be propositionally equal.

One promising solution to the problem of function extensionality in a constructive setting is homotopy type theory [14] 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. In some proofs, we assume to be working in a homotopy type theory setting were function extensionality and univalence hold. We made this choice for convenience reasons as most of our proofs were first written without using univalence. In our AGDA development, we use the flag `--without-K` which disable the K-rule during pattern-matching. As far as we can tell the only modules using the K-rule are only using it for the set of natural numbers (`Fin` and `Vec`). This should be resolved as soon as `--without-K` becomes smarter about types which are sets.

2. Folds and explorations

In order to be able to work more conveniently with parametricity later on, we focus here only on `foldMap` after it has already been applied to a container. Since we extensively use this type we give it a name, `Explore` which is expressed in HASKELL as:

```
type Explore a = ∀ m. Monoid m ⇒ (a → m) → m

toExplore :: ∀ a t. Foldable t ⇒ t a → Explore a
toExplore t f = foldMap f t
```

The AGDA version of `Explore` is given below, instead of a type class constraint, the monoid operations are passed explicitly:

Definition 1. *An exploration function for a type A is given a type M, a value ϵ of type M, a function $\_\oplus\_$ of type $M \to M \to M$, and function f of type $A \to M$. The exploration function finally yields a result of type M:*

```
Explore : ★ → ★
Explore A = {M : ★}(ε : M)(_⊕_ : M → M → M)
          → (A → M) → M
```

For any type `A`, an exploration function is given a default result ϵ, a binary operator $\_\oplus\_$ 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. One viewpoint is that the task of an exploration function is thus to transform any small operator `_⊕_` into the corresponding big operator $\bigoplus$ of type `(A → M) → M`. For instance, if `explore` is an exploration function for a type `A`, then `explore 0 _+_` is $\sum$ and `explore 1 _*_` is $\prod$, where `0`, `1`, `_+_` and `_*_` are defined on the type `ℕ`.

A continuation monad with environment: The type of exploration can be viewed as a continuation monad (`(A → M) → M`), with two reader monad transformers giving access to `ε` and `_⊕_`.

Monoid laws: Note that the type does not specify that the exploration will be over a monoid. The laws are not given, only the operations. When proving properties about explorations, the monoid laws will have to be assumed as well. Not having to provide the monoid laws makes it easier to write transformations of exploration functions.

Finiteness: Given AGDA's type discipline, the type `Explore A` enforces that any exploration function will only explore a finite number of values of the type `A`. This is enforced by AGDA functions being total (strongly normalizing and exhaustively defined) and by parametricity [2, 13, 16]: since the exploration function knows nothing about the type `M` it must use what is given to it.

Exhaustiveness: 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. The exhaustiveness of an exploration implies the finiteness of `A`.

2.1 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`, the type of six sided dice:

```
data D6 : ★ where
  ⚀ ⚁ ⚂ ⚃ ⚄ ⚅ : D6

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` respectively. Figure 1 shows the function `empty-explore`, 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 `𝟘`, `𝟙` and `𝟚`. For sum types `A ⊎ B`, the exploration `explore⊎ eᴬ eᴮ ε _⊕_ f` combines the two results given by exploring the function `f` specialised to types `A` and `B` using `inl` and `inr` — the injections for the type `_⊎_`. The two results are then combined using `_⊕_`. For Cartesian products `A × B`, the exploration `explore× eᴬ eᴮ ε _⊕_ f` nests the exploration of `B` into the function exploring `A`. Note how this combinator is independent of the operator `_⊕_`. Support for dependent pairs and functions is detailed in Section 4.

```
empty-explore : ∀ {A} → Explore A
empty-explore 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 ∘ inl)) ⊕ (eᴮ ε _⊕_ (f ∘ inr))

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)))

explore𝟘 : Explore 𝟘
explore𝟘 = empty-explore

explore𝟙 : Explore 𝟙
explore𝟙 = point-explore 0₁

explore𝟚 : Explore 𝟚
explore𝟚 = merge-explore (point-explore 0₂)
                         (point-explore 1₂)
```

Figure 1. Exploration functions

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

Derived big operators: In Figure 2 we recall some standard big operators. These are derived from any exploration function by choosing the appropriate monoid structure. Sums and products are defined using the monoids `(ℕ , 0 , _+_)` and `(ℕ , 1 , _*_)` as mentionned earlier. From a summation function we derive a function `count` which is counting the occurrences of a given predicate. Summing (with `sum` or `count`) using a constant function `1` yields the size of the exploration. Finally the functions `all` and `any` test a given predicate to tell whether it holds for all the explored values or one of the explored values, respectively.

2.2 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 use the term exploration transformer for the operations which map exploration functions to exploration functions.

A prototypical program involving an exploration function is the brute force exhaustive search. This could be the search to inverse

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

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

  count : (A → 𝟚) → ℕ
  count f = sum (𝟚▷ℕ ∘ f)
            {- 𝟚▷ℕ maps 0₂ to 0 and 1₂ to 1 -}

  size : ℕ
  size = count (const 1₂)

  all : (A → 𝟚) → 𝟚
  all = exploreᴬ 1₂ _∧_

  any : (A → 𝟚) → 𝟚
  any = exploreᴬ 0₂ _∨_

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

  tree : Tree A
  tree = exploreᴬ empty fork leaf

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

Figure 2. Derived big operators

a function, such as a hashing function. Sometimes the domain (message space) is relatively small and searching it can be used to gather information. Here let us suppose a type A together with an exploration function exploreᴬ, a type B together with an equality test (_==_ has type B → B → 𝟚), and a function H : A → B. In practice one might think of the function H as being hard to inverse. The following program naively inverts H by exploring all possible messages, and returning the list of all messages which maps to the input digest:

```
H⁻¹-list : B → List A
H⁻¹-list b = exploreᴬ [] _++_ λ a →
   if H a == b then [ a ] else []
```

While straightforward, the exploration in H⁻¹-list shows a lack of modularity: indeed the data structure (here a list) for the result 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
(eᴬ >>= eᴮ) ε _⊕_ f = eᴬ ε _⊕_ λ x →
                                eᴮ x ε _⊕_ f
```

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

The function gfilter-explore (for generic filter) discards undesirable values and 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
gfilter-explore f eᴬ = eᴬ >>= λ x → case (f x) of λ
                    { nothing  → empty-explore
                    ; (just y) → point-explore y }

filter-explore  : (p : A → 𝟚)
                → Explore A → Explore A
filter-explore p = gfilter-explore (λ x →
 if p x then just x else nothing)
```

The previous example, inverting a function H, can be built using filter-explore (λ a → H a == b); the result is then an exploration from which one can get a list (using the list monoid) or the first matching values (using a monoid for Maybe).

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 [4]) yield exploration transformers.

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

As a last example of a 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 M → M. The body composes the original small operator _⊕_ 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. A proof of this technique has been given in [15] and it is our Corollary 2. Notice that this technique is nicely captured by the following exploration transformer:

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

3. Relational Parametricity

Since the type of foldMap is polymorphic it satisfies some theorems for free [16]. Indeed some programming languages have been shown to enjoy a so called abstraction theorem [2, 13, 16]. The theory behind HASKELL and AGDA are known to enjoy this ab-

[4] flip transforms a two arguments function to flip its argument.

```
⟦Explore⟧ : Explore A → Explore A → ★
⟦Explore⟧ e e' = (M M' : ★)(⟦M⟧ : M → M' → ★)
  (ε : M)(ε' : M')(εᵣ : ⟦M⟧ ε ε')
  (_⊕_ : M → M → M)(_⊕'_ : M' → M' → M')
  (⊕ᵣ : ∀ {x y}{x' y'} → ⟦M⟧ x x' → ⟦M⟧ y y'
       → ⟦M⟧ (x ⊕ y) (x' ⊕ y'))
  (f : A → M)(f' : A → M')
  (fᵣ : ∀ x → ⟦M⟧ (f x) (f' x))
  → ⟦M⟧ (e ε _⊕_ f) (e' ε' _⊕'_ f')
```

Figure 3. Parametricity relation for Explore

straction theorem. The statement for such free-theorems are mechanically derived from types. Any well-typed program enjoys the free-theorem arising from its type. While they are uninformative for monomorphic types they are interesting for polymorphic types. Usually, these theorems are stated using pen and paper proofs for HASKELL programs but if we move to a dependently typed language, such as AGDA, the types, programs, statements and proofs can inhabit a common system. Although these free-theorems are mechanical they are currently not automated by the system. In our online development we provide and use a library which helps streamline this process, we however here present a more syntactic approach.

The high level overview is that each type T : ★ will induce a (binary) relation, which we will denote by oxford brackets ⟦T⟧ : T → T → ★. The (binary) free-theorem, also known as the fundamental theorem, is that this relation is reflexive, i.e for all terms t : T there is a proof term ⟦t⟧ : ⟦T⟧ t t. If parametricity was internalised then this proof would come for free, but here we instead need to prove it for each instance. The ⟦_⟧ relation is defined by induction on the type. For example, functions are in the relation if they map related inputs to related outputs:

```
⟦A→B⟧ : (A → B) → (A → B) → ★
⟦A→B⟧ f f' = (x x' : A)
  → ⟦A⟧ x x' → ⟦B⟧ (f x) (f' x')
```

Since polymorphism is expressed using a universe type ★, we need to know what the relation ⟦★⟧ is. Following [2] we pick ⟦★⟧ to be the type of all relations. Intuitively ⟦★⟧ should at least contain the identity relation since it corresponds to the generated relation for basics types such as 𝟚. The main reason to include all relations is to strengthen the parametricity results. Indeed when using the parametricity result of a polymorphic function one get to choose freely the relation which is quite useful.

```
⟦★⟧ : ★ → ★ → ★
⟦★⟧ A B = A → B → ★
```

Furthermore we need to extend the relation on functions to dependent functions in order to express the type of polymorphic functions. We follow [2] again and the tricky part in defining the relation for a dependent function such as (x : A) → B x is that the type of B is B : A → ★. The relation on B must have the type ⟦A→★⟧ B B, namely (x x' : A) → ⟦A⟧ x x' → B x → B x' → ★. This means that the relation on B is indexed on the two A which are known to be related by ⟦A⟧.

```
⟦ΠAB⟧ : ((x : A) → B x) → ((x : A) → B x) → ★
⟦ΠAB⟧ f f' = (x x' : A)
  → (xᵣ : ⟦A⟧ x x') → ⟦B⟧ x x' xᵣ (f x) (f' x')
```

Now all the tools are available to derive what the relation is for the Explore type. This relation is defined in Figure 3, and while it looks daunting it is fairly straightforward to use. The trick lies in that it is possible to pick any relation for ⟦M⟧. For example we use it to prove that a monoid homomorphism distributes over explore.

Theorem 1. *For any type A, exploration function e^A : Explore A, two monoids: (M , ε , _⊕_) and (N , ι , _⊗_), we have a monoid homomorphism h from M to N, and a function f : A → M, then h (e^A ε _⊕_ f) ≡ e^A ι _⊗_ (h ∘ f)*

Proof. By parametricity of e^A we pick ⟦M⟧ x y to be h x ≡ y. We need to prove: h ε ≡ ι and for all x, x', y and y' such that h x ≡ x', h y ≡ y we have h (x ⊕ y) ≡ x' ⊗ y'. Both of these requirements follow from the fact that h is a monoid homomorphism. The final requirement is that for all x, h (f x) ≡ h (f x) holds, which is trivial. □

Corollary 1. *For any type A, exploration function e^A : Explore A, function f : A → ℕ and constant k : ℕ, we have k ∗ sum e^A f ≡ sum e^A (λ x → k ∗ f x).*

Proof. By Theorem 1 and the fact that (_∗_ k) is a monoid homomorphism, since k ∗ 0 ≡ 0 and k ∗ (x + y) ≡ k ∗ x + k ∗ y. □

Theorem 2. *For any type A, exploration function e^A : Explore A, a monoid (M , ε , _⊕_) equipped with a preorder _≤_ such that _⊕_ is monotonic, two functions f , g : A → M such that for all x f x ≤ g x, we have e^A ε _⊕_ f ≤ e^A ε _⊕_ g.*

Proof. By parametricity of e^A we pick ⟦M⟧ to be _≤_, all the requirements follow from assumptions. □

We remark that nowhere in our AGDA development we postulate these parametricity results. Instead, for each exploration function we provide the corresponding proof of the parametricity result. Building such a proof is mechanical thanks to the functional nature of the underlying abstraction theorem.

3.1 Exploration Principle

The parametricity relation is a powerfull tool but sometimes we want something closer to an induction principle. An induction principle allows the target proprety (known as ⟦M⟧ in our previous proofs) to be not only a relation between two explorations, but can be an arbitrary predicate on the exploration function itself.

Definition 2. *The exploration principle states that any property P on an exploration function e^A holds if: P holds for empty-explore; P holds for all points (using point-explore); and P is preserved by merge-explore.*

```
Exploreᴾ : ∀ {A} → Explore A → ★
Exploreᴾ {A} eᴬ =
  ∀ (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 eᴬ
```

Proper exploration functions come with the principle defined above. This principle is the induction principle on binary trees where empty, fork, 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, forks are calls to $\_\oplus\_$, and leaves are calls to `f`.

Moreover, while the type of the principle (i.e. $\mathtt{Explore}^{\mathtt{P}}$) also 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 $\mathtt{exploreD6}^{\mathtt{P}}$ is almost like `exploreD6`:

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

In an impredicative setting at least (such as $\star : \star$), the principle is equivalent to the parametricity relation, but so far we have not been able to prove this correspondence in a predicative setting. This causes some duplication in the amount of work one has to do when providing an exploration function, though this work is mostly mechanical.

Theorem 3. *For any type A, exploration function $e^A : \mathit{Explore}\ A$, a commutative monoid $(M\ ,\ \epsilon\ ,\ \_\oplus\_)$ and two functions $f\ ,\ g : A \to M$, we have $e^A\ \epsilon\ \_\oplus\_\ (\lambda\ x \to f\ x \oplus g\ x) \equiv e^A\ \epsilon\ \_\oplus\_\ f \oplus e^A\ \epsilon\ \_\oplus\_\ g$.*

Proof. By the principle of $\mathtt{e}^{\mathtt{A}}$ and picking the motive [5] $\mathtt{P\ e}$ to be $\mathtt{e}\ \epsilon\ \_\oplus\_\ (\lambda\ \mathtt{x} \to \mathtt{f\ x} \oplus \mathtt{g\ x}) \equiv \mathtt{e}\ \epsilon\ \_\oplus\_\ \mathtt{f} \oplus \mathtt{e}\ \epsilon\ \_\oplus\_\ \mathtt{g}$. We need to show `P empty-explore` which is $\epsilon \equiv \epsilon \oplus \epsilon$ which follows by monoid law. The case $\mathtt{P}\ (\mathtt{merge\text{-}explore}\ \mathtt{e}_0\ \mathtt{e}_1)$ where $\mathtt{P\ e}_0$ and $\mathtt{P\ e}_1$ follows by the interchange law (i.e for all $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, and $\mathtt{d}$ then $(\mathtt{a} \oplus \mathtt{b}) \oplus (\mathtt{c} \oplus \mathtt{d}) \equiv (\mathtt{a} \oplus \mathtt{c}) \oplus (\mathtt{b} \oplus \mathtt{d})$). Finally we need to prove for all $\mathtt{x}$ that $\mathtt{P}\ (\mathtt{point\text{-}explore\ x})$ which is $\mathtt{f\ x} \oplus \mathtt{g\ x} \equiv \mathtt{f\ x} \oplus \mathtt{g\ x}$ which is trivial. □

Theorem 4. *For any type A, exploration function $e^A : \mathit{Explore}\ A$, two monoids[6] $(M\ ,\ \epsilon_m\ ,\ \_\oplus_m\_)$ and $(N\ ,\ \epsilon_n\ ,\ \_\oplus_n\_)$, two functions $f_m : A \to M$ and $f_n : A \to N$, we have the exploration of the product monoid is the product of explorations, namely $e^A\ \epsilon\ \_\oplus\_ < f_m \times f_n > \equiv (e^A\ \epsilon_m\ \_\oplus_m\_\ f_m\ ,\ e^A\ \epsilon_n\ \_\oplus_n\_\ f_n)$ where $((M \times N)\ ,\ \epsilon\ ,\ \_\oplus\_)$ is the product monoid.*

Proof. By the principle of $\mathtt{e}^{\mathtt{A}}$ and picking the motive to be $\mathtt{P\ e}$ to be $\mathtt{e}\ \epsilon\ \_\oplus\_ < \mathtt{f}_m \times \mathtt{f}_n > \equiv (\mathtt{e}\ \epsilon_m\ \_\oplus_m\_\ \mathtt{f}_m\ ,\ \mathtt{e}\ \epsilon_n\ \_\oplus_n\_\ \mathtt{f}_n)$, we need to show `P empty-explore` which holds by definition of the product monoid. The case for $\mathtt{P}\ (\mathtt{merge\text{-}explore}\ \mathtt{e}_0\ \mathtt{e}_1)$ where $\mathtt{P\ e}_0$ and $\mathtt{P\ e}_1$ follows by congruence of $\_\oplus\_$ and its definition. Finally we need to prove for all $\mathtt{x}_m$ and $\mathtt{x}_n$ that $\mathtt{P}\ (\mathtt{point\text{-}explore}\ (\mathtt{x}_m\ ,\ \mathtt{x}_n))$ which is $< \mathtt{f}_m \times \mathtt{f}_n > (\mathtt{x}_m\ ,\ \mathtt{x}_n) \equiv (\mathtt{f}_m\ \mathtt{x}_m\ ,\ \mathtt{f}_n\ \mathtt{x}_n)$ which holds by definition. □

We can now prove how `explore-endo` enables to re-associate an exploration.

Theorem 5. *For any type A, exploration function $e^A : \mathit{Explore}\ A$, monoid $(M\ ,\ \epsilon\ ,\ \_\oplus\_)$, function $f : A \to M$, and point $z : M$, we have $e^A\ \epsilon\ \_\oplus\_\ f \oplus z \equiv e^A\ \mathit{id}\ \_\circ\_\ (\_\oplus\_ \circ f)\ z$.*

Proof. By the principle of $\mathtt{e}^{\mathtt{A}}$ and picking the motive $\mathtt{P\ e}$ to be $\forall\ \mathtt{z} \to \mathtt{e}\ \epsilon\ \_\oplus\_\ \mathtt{f} \oplus \mathtt{z} \equiv \mathtt{e\ id}\ \_\circ\_\ (\_\oplus\_ \circ \mathtt{f})\ \mathtt{z}$, we need to show `P empty-explore` which is $\forall\ \mathtt{z} \to \epsilon \oplus \mathtt{z} \equiv \mathtt{z}$ and follows by monoid law. The case for $\mathtt{P}\ (\mathtt{point\text{-}explore\ x})$ holds by definition. Finally we need to prove the case for $\mathtt{P}\ (\mathtt{merge\text{-}explore}\ \mathtt{e}_0\ \mathtt{e}_1)$ where $\mathtt{P\ e}_0$ and $\mathtt{P\ e}_1$ hold. By definition it amounts to proving that for all $\mathtt{z}$, $(\mathtt{e}_0\ \epsilon\ \_\oplus\_\ \mathtt{f} \oplus \mathtt{e}_1\ \epsilon\ \_\oplus\_\ \mathtt{f}) \oplus \mathtt{z}$ equals $\mathtt{e}_0\ \mathtt{id}\ \_\circ\_\ (\_\oplus\_ \circ \mathtt{f})\ (\mathtt{e}_1\ \mathtt{id}\ \_\circ\_\ (\_\oplus\_ \circ \mathtt{f})\ \mathtt{z})$. The assumption $\mathtt{P\ e}_1$ can be used on $\mathtt{z}$ and $\mathtt{P\ e}_0$ can be used on $\mathtt{e}_1\ \epsilon\ \_\oplus\_\ \mathtt{f} \oplus \mathtt{z}$. Using the associativity and congruence of $\_\oplus\_$ the proof is complete. □

Corollary 2. *For any type A, exploration function $e^A : \mathit{Explore}\ A$, then any exploration can be re-associated using the monoid on endomorphisms, namely for all monoid $(M\ ,\ \epsilon\ ,\ \_\oplus\_)$ and function $f : A \to M$, we have $e^A\ \epsilon\ \_\oplus\_\ f \equiv \mathit{explore\text{-}endo}\ e^A\ \epsilon\ \_\oplus\_\ f$.*

Proof. Use Theorem 5 with $\mathtt{z}$ being ϵ and conclude by monoid laws. □

Exploration functions can be concretised to binary trees[7]. Binary trees can be explored using the fold function for trees. This allows us to treat exploration functions as data.

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

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

toTree : ∀ {A} → Explore A → Tree A
toTree eᴬ = eᴬ empty fork leaf
```

Theorem 6. *For any type A, exploration function $e^A : \mathit{Explore}\ A$, monoid $(M\ ,\ \epsilon\ ,\ \_\oplus\_)$ and function $f : A \to M$, we have $e^A\ \epsilon\ \_\oplus\_\ f \equiv \mathit{foldMap}^T\ (\mathit{toTree}\ e^A)\ \epsilon\ \_\oplus\_\ f$.*

Proof. By the principle of $\mathtt{e}^{\mathtt{A}}$ and picking the motive to be $\mathtt{P\ e} = \mathtt{e}\ \epsilon\ \_\oplus\_\ \mathtt{f} \equiv \mathtt{foldMap}^{\mathtt{T}}\ (\mathtt{toTree\ e})\ \epsilon\ \_\oplus\_\ \mathtt{f}$. All cases are trivial. □

[5] It is common to refer as `P` being the motive for the induction which is a form of elimination. As Conor McBride writes in [10] "we should give elimination a *motive*".

[6] The monoid laws are actually not used for this theorem.

[7] Binary trees do not form a monoid with strict equality but our exploration functions do not require it either.

4. Exploration and dependent types

Big operators over types: Intuitively Σ is the big operator for `_⊎_` and Π the big operator for `_×_`. For any type `A` and e^A `: Explore A`, the monoids `(★ , 𝟘 , _⊎_)` and `(★ , 𝟙 , _×_)` can be used to compute a type from the explored values of `A`. We call these operators Σ^e and Π^e:

```
Σ^e : (A → ★) → ★
Σ^e = e^A 𝟘 _⊎_

Π^e : (A → ★) → ★
Π^e = e^A 𝟙 _×_
```

For any type family `B`, a value of type Σ^e `B` is a composition of injections (`inl`/`inr`) until reaching a value of type `B x` for some `x : A`. Similarly, a value of type Π^e `B` is a tuple of nested pairs storing a value `B x` for each `x` explored.

When all the values of type `A` are exhaustively and uniquely explored, then the type operators Σ^e and Π^e are equivalent to `Σ A` and `Π A` respectively. When it is so, Σ^e and Π^e are said to be adequate Σ-type and Π-type.

```
Adequate-Σ : ((A → ★) → ★) → ★
Adequate-Σ Σ^A = ∀ F → Σ^A F ≡ Σ A F

Adequate-Π : ((A → ★) → ★) → ★
Adequate-Π Π^A = ∀ F → Π^A F ≡ Π A F
```

These type operators (Σ^e and Π^e) can be read logically as finitary qualifiers ($\exists$ and $\forall$).

Small scale reflection by exhaustive testing:

Theorem 7. *For any type `A`, exploration function e^A `: Explore A`, and function `f : A → 𝟚`, assuming furthermore that the derived Π^e is adequate, then `all` e^A `f` returns 1_2 exactly when `f` returns 1_2 for all `x` of type `A`, namely `✓ (all` e^A `f) ≡ ∀ x → ✓ (f x)` where `✓` maps `𝟚` to `★` and the function `all` is defined in Figure 2.*

Proof. Since `✓` forms a monoid homomorphism from `(𝟚 ,` 1_2 `, _∧_)` to `(★ , 𝟙 , _×_)` one can distribute this homomorphism using Theorem 1. Remains to show that Π^e e^A `(✓ ∘ f)` is equal to `∀ x → ✓ (f x)` which holds by adequacy of Π^e. □

This theorem can be used to provide a generic tool for proofs by reflection. This tool can be used to prove any statement for which the domain is amenable to exhaustive testing. The function `check!` below takes any predicate `f` on `A` expressible as a function to `𝟚`, the second argument is implicit and thus forces the type checker to normalise the expression `all` e^A `f`. If this expression normalise to 0_2 then the type checker fails to find a term of type `𝟘`, if it normalise to 1_2 then the type checker can apply the η-rule for the type `𝟙` to establish the existence and uniqueness of the implicit argument. The function `check!` then returns a proof that `f x` is 1_2 for all `x`. Internally the function `check!` uses Theorem 7 in the forward direction.

```
check! : ∀ f → {pf : ✓ (all e^A f)}
           → (∀ x → ✓ (f x))
```

As an example of the use of `check!` we automatically derive a proof of the distributivity of `_∧_` over `_∨_`. One first define the property as single function over `𝟚`, no currying is used here. Then from the exploration function on `𝟚 × 𝟚 × 𝟚` one get the `check!` which proves the goal by certified exhaustive search.

```
prop-∧-∨-distr (x , y , z) =
  x ∧ (y ∨ z) == x ∧ y ∨ x ∧ z

check-∧-∨-distr : ∀ x y z →
  ✓(x ∧ (y ∨ z) == x ∧ y ∨ x ∧ z)
check-∧-∨-distr x y z =
  check! prop-∧-∨-distr (x , y , z)
```

Similarly, the disjunction is related to Σ-types, but not with a bi-implication and not an equivalence. Indeed, `✓(x ∨ y)` has at most one inhabitant while `✓ x ⊎ ✓ y` has at most two.

Theorem 8. *For any type `A`, exploration function e^A `: Explore A`, and function `f : A → 𝟚`, assuming furthermore that the derived Σ^e is adequate, then `any` e^A `f` returns 1_2 exactly when `f` returns 1_2 for some `x` of type `A`, namely `✓ (any` e^A `f) ↔ ∃ λ x → ✓ (f x)` where the function `any` is defined in Figure 2.*

Proof. The proof is similar to Theorem 7 except that the monoid `(★ , 𝟘 , _⊎_)` is choosen up to bi-implication instead of equality. □

4.1 Explorable types are decidable

When working with finite types it is possible to appeal to classical logic principles. Using exploration functions we can for example recover decidability for Σ and Π-types. We recall that `Dec A` is equivalent `A ⊎ ¬ A`.

Lemma 1. *Decidability is provable for types `𝟘`, `𝟙` and is closed under sums and products (`_⊎_`, `_×_`).*

These proofs are straightforward and as they are available in our online development, we omit them for concisions.

Let `B : A → ★` be a type family, we call `B` as a decidable predicate if and only if `B x` is decidable for all `x : A`.

Theorem 9. *Let `A` be a type, e^A be an exploration on `A`, and Π^e e^A be adequate. If `B` is a decidable predicate then `Π A B` is decidable.*

Proof. We start by proving that Π^e e^A `B` is decidable. Using induction on e^A with motive `P e` to be `Dec (`Π^e `e B)`. We need to show that `P empty-explore` holds, which is `Dec 𝟙` and follows from Lemma 1. We need to show that `P (merge-explore` e_0 e_1`)` holds assuming `P` e_0 and `P` e_1, it follows from `Dec` being closed under products `_×_`. We need to show that `P (point-explore x)` holds, which is `Dec (B x)` which holds since `B` is a decidable predicate. Finally one uses the adequacy of Π^e e^A to conclude the proof. □

Theorem 10. *Let `A` be a type, e^A be an exploration on `A`, and Σ^e e^A be adequate. If `B` is a decidable predicate then `Σ A B` is decidable.*

Proof. The proof follows the same structure as for Π-types (Lemma 9), where `Dec 𝟘` is the base case and the merge case uses the fact that `Dec` is closed under sums. □

Exploring Σ-types: The function `explore×` in Figure 1 explores the cartesian product `A × B` given explorations for `A` and `B`. This construction nicely scales to dependent pairs. To explore `Σ A B` one needs a family of explorations for each `B x` where `x` has type `A`.

This implies a single change in comparison to `explore×`, namely `x` is given to $\texttt{explore}^{\texttt{B}}$:

```
exploreΣ : Explore A → (∀ x → Explore (B x))
         → Explore (Σ A B)
exploreΣ explore^A explore^B ε _⊕_ f
   = explore^A ε _⊕_ λ x →
       explore^B x ε _⊕_ λ y →
         f (x , y)
```

The relational parametricity applies to the definition of `exploreΣ`. While not yet fully automated in AGDA, the mechanical aspect of parametricity is appreciated even when dealing with types and programs as short as `Explore`, Σ, and `exploreΣ`. The definitions for $[\![\Sigma]\!]$, $[\![\texttt{explore}\Sigma]\!]$, and $\texttt{explore}\Sigma^{\texttt{P}}$ are given in appendix A.2.

Exploration functions for exploring functions: Exploring a function type (such as `Explore (A → B)`), would combine functions $\texttt{f}_0\ \texttt{f}_1\ \dots\ \texttt{f}_n$ `: A → B` using the provided binary operator. Ideally one would combine information about `A` and `B` to explore `A → B`. While we found no way to directly exhaustively explore functions there is an attractive workaround: one can use type equivalences on functions to incrementally build such an exploration function. Namely, one decomposes the domain with type equivalences towards simpler types we can explore:

$$
\begin{array}{lcl}
(\texttt{A} \uplus \texttt{B}) \to \texttt{C} & \simeq & (\texttt{A} \to \texttt{C}) \times (\texttt{B} \to \texttt{C}) \\
(\texttt{A} \times \texttt{B}) \to \texttt{C} & \simeq & \texttt{A} \to (\texttt{B} \to \texttt{C})
\end{array}
$$

These type equivalences require function 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 type equivalences are required to prove their adequacy.

5. Sums, products and type equivalences

5.1 Adequate sums and products

Our original motivation was to work with summation functions as a way to compute and reason about uniform discrete probability distributions. Using an exploration function, we can derive a summation function which has stronger properties (the free-theorems discussed before), we can then sum the events over all the values of a given type. Exploring types more than containers is illustrated in Section 5.2 where we model probabilistic functions as deterministic functions with an extra argument for the randomness.

In this part we develop adequate summations and products and some properties they enjoy. In Section 5.2 we build on summations and show how our model for probabilistic functions yield uniform and discrete probability distributions. In particular probabilistic equivalence is equivalent to type equivalence in Corollary 4. This corollary follows from Theorem 12 and Theorem 16 developed in this section.

How can we ensure that we have a correct summation function? We need to ensure that an adequate summation function is going to count every value exactly once (i.e an adequate 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 sizes[8] 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 `Σ A F` is acting as a big operator for disjoint union of all `F x` where `x` is of type `A`. Therefore the size of a Σ-type is the summation of the sizes over the type family: $\sharp\,(\texttt{Fin n}) \equiv \texttt{n}$ and $\sharp\,(\Sigma\ \texttt{A}\ \texttt{F}) \equiv \sum_{\texttt{x}\in\texttt{A}} \sharp\,(\texttt{F x})$.

[8] We use the notion of size only as an informal guide.

Using these size relations we can show that $\texttt{sum}^{\texttt{A}}$ a summation function is correct, assuming a particular type equivalence exists. Since type equivalences preserve sizes, we argue as follows.

$$
\begin{array}{lclcl}
\texttt{sum}^{\texttt{A}}\ \texttt{f} & \equiv & \sharp\left(\texttt{Fin}\ (\texttt{sum}^{\texttt{A}}\ \texttt{f})\right) & \equiv & \sharp\left(\Sigma\ \texttt{A}\ (\texttt{Fin} \circ \texttt{f})\right) \\
 & \equiv & \sum_{\texttt{x}\in\texttt{A}} \sharp\,(\texttt{Fin}\ (\texttt{f x})) & \equiv & \sum_{\texttt{x}\in\texttt{A}} \texttt{f x}
\end{array}
$$

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 equivalence between $\Sigma\ A\ (Fin \circ f)$ and $Fin\ (sum^{A}\ f)$. In* AGDA*: $Adequate\text{-}sum\ sum^{A} = \forall\ f \to \Sigma\ A\ (Fin \circ f) \simeq Fin\ (sum^{A}\ f)$.*

This correspondence can be further extended to products, as 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:

$$
\begin{array}{lclcl}
\texttt{prod}^{\texttt{A}}\ \texttt{f} & \equiv & \sharp\left(\texttt{Fin}\ (\texttt{prod}^{\texttt{A}}\ \texttt{f})\right) & \equiv & \sharp\left(\Pi\ \texttt{A}\ (\texttt{Fin} \circ \texttt{f})\right) \\
 & \equiv & \prod_{\texttt{x}\in\texttt{A}} \sharp\,(\texttt{Fin}\ (\texttt{f x})) & \equiv & \prod_{\texttt{x}\in\texttt{A}} \texttt{f x}
\end{array}
$$

Definition 4. *A function $prod^{A}$ for a type A is said to be an adequate product if for all f there is an equivalence between $\Pi\ A\ (Fin \circ f)$ and $Fin\ (prod^{A}\ f)$.*
In AGDA*: $Adequate\text{-}product\ prod^{A} = \forall\ f \to \Pi\ A\ (Fin \circ f) \simeq Fin\ (prod^{A}\ f)$.*

Our first use of adequacy for sums and products is to prove the following equation:

$$
\forall (f \in (A \times B) \to \mathbb{N}),\ \prod_{x\in A} \sum_{y\in B} f(x,y) \equiv \sum_{g\in(A\to B)} \prod_{x\in A} f(x,g(x))
$$

At first we prove a more general result, where `B` is a family indexed by `A` and thus dependent functions and dependent pairs are required. The non-dependent version is given as a corollary.

Theorem 11. *Let $prod^{A}$ be an adequate product function for the type A. Let $sum^{A\,B}$ be an adequate summation function for a type $\Pi\ A\ B$. Finally let sum^{B} be a family over A of summation functions on the type B. Then for all function $f\ :\ (x\ :\ A) \to B\ x \to \mathbb{N}$, $prod^{A}\ (\lambda\ x \to sum^{B}\ x\ (\lambda\ y \to f\ x\ y))$ is equal to $sum^{A\,B}\ (\lambda\ g \to prod^{A}\ (\lambda\ x \to f\ x\ (g\ x)))$.*

Proof. Using the adequacy properties together with the type equivalence between $\Pi\ \texttt{A}\ (\lambda\ \texttt{x} \to \Sigma\ (\texttt{B x})\ \lambda\ \texttt{y} \to \texttt{C x y})$ and $\Sigma\ (\Pi\ \texttt{A B})\ \lambda\ \texttt{f} \to \Pi\ \texttt{A}\ \lambda\ \texttt{x} \to \texttt{C x}\ (\texttt{f x})$. The logical interpretation of the forward direction is usually known as the dependent axiom of choice. Categorically a map into a product (Σ-type) is a product of maps. □

Corollary 3. *Let sum^{B} and $sum^{A\,B}$ be adequate summation functions for a type B and $A \to B$ respectively. Furthermore let $prod^{A}$ be an adequate product function for the type A. Then for all function $f\ :\ A \to B \to \mathbb{N}$, $prod^{A}\ (\lambda\ x \to sum^{B}\ (\lambda\ y \to f\ x\ y))$ is equal to $sum^{A\,B}\ (\lambda\ g \to prod^{A}\ (\lambda\ x \to f\ x\ (g\ x)))$.*

Proof. Since non-dependent functions are a particular case of dependent functions one can directly use Theorem 11. □

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

Lemma 2. *Having an adequate summation function* sum *over type* A, *with a derived size* $\mathit{size} : \mathbb{N}$, *it is possible to construct an equivalence* $\mathit{Fin}\ \mathit{size} \simeq A$.

Proof. By $\mathtt{sum}$ being adequate summation and the type equivalence $\Sigma\ \mathtt{A}\ (\lambda\ \_ \to \mathtt{Fin}\ 1) \simeq \mathtt{A}$. □

Lemma 3. *Let* A_0, A_1 *be types, and* $A{=}$ *be a type identity :* $A_0 \equiv A_1$. *Let* B_1, B_1 *be type families on* A_0 *and* A_1 *respectively. Let* $B{=}$ *a family over* A_0 *of equivalences between* B_0 *and* B_1. *The type of* $B{=}$ *is* $(x : A_0) \to B_0\ x \equiv B_1\ ((\mathit{coe}\ A{=})\ x)$ *and* $\mathit{coe}\ A{=}$ *is the identity transport along* $A{=}$. *It is then possible to construct a path* $\Sigma\ A_0\ B_0 \equiv \Sigma\ A_1\ B_1$.

Proof. By based path induction on $\mathtt{A{=}}$ one has only to consider the case for reflexivity on the base point $\mathtt{A_0}$. Notice that the family $\mathtt{B_1}$ now is on $\mathtt{A_0}$, and that $\mathtt{B{=}}$ is now convertible to $(\mathtt{x} : \mathtt{A_0}) \to \mathtt{B_0}\ \mathtt{x} \equiv \mathtt{B_1}\ \mathtt{x}$ since $\mathtt{coe}$ computes to the identity function on the reflexivity path. It remains to show a path between $\Sigma\ \mathtt{A_0}\ \mathtt{B_0}$ and $\Sigma\ \mathtt{A_1}\ \mathtt{B_1}$, which amounts to first use function extensionality on $\mathtt{B{=}}$ to get a path $\mathtt{B_0} \equiv \mathtt{B_1}$, which can then be applied to the context $\Sigma\ \mathtt{A_0}$. □

Lemma 4. *Let* A_0, A_1 *be types, and* $A{\simeq}$ *be a type equivalence :* $A_0 \simeq A_1$. *Let* B_1, B_1 *be type families on* A_0 *and* A_1 *respectively. Let* $B{=}$ *a family over* A_0 *of paths between* B_0 *and* B_1. *The type of* $B{=}$ *is* $(x : A_0) \to B_0\ x \equiv B_1\ (\cdot\!\!\to A{\simeq}\ x)$ *and* $\cdot\!\!\to A{\simeq}$ *projects the* $A_0 \to A_1$ *function. It is then possible to construct an equivalence* $\Sigma\ A_0\ B_0 \simeq \Sigma\ A_1\ B_1$.

Proof. The equivalence $\mathtt{A{\simeq}}$ is transformed into a path using the univalence axiom $\mathtt{ua}$. To use the previous Lemma 3 it remains to show a family of paths: $\forall\ \mathtt{x} \to \mathtt{B_0}\ \mathtt{x} \equiv \mathtt{B_1}\ (\mathtt{coe}\ (\mathtt{ua}\ \mathtt{A{\simeq}})\ \mathtt{x})$. Considering such an $\mathtt{x} : \mathtt{A_0}$ we first use the path $(\mathtt{B{=}})\ \mathtt{x}$. To show a path between $\mathtt{B_1}\ (\cdot\!\!\to \mathtt{A{\simeq}}\ \mathtt{x})$ and $\mathtt{B_1}\ (\mathtt{coe}\ (\mathtt{ua}\ \mathtt{A{\simeq}})\ \mathtt{x})$ we apply the context $\mathtt{B_1}$. Finally we use the β-rule for the univalence axiom which gives a path between $\cdot\!\!\to \mathtt{A{\simeq}}\ \mathtt{x}$ and $\mathtt{coe}\ (\mathtt{ua}\ \mathtt{A{\simeq}})\ \mathtt{x}$, which concludes the proof. □

Theorem 12. *Given two adequate summation functions* sum^A *and* sum^B *for types* A *and* B *respectively, for all equivalences* $\pi : A \simeq B$ *and functions* $f : B \to \mathbb{N}$ *the summation* $\mathit{sum}^A\ (f \circ \pi)$ *is equal to the summation* $\mathit{sum}^B\ f$.

Proof. Using adequacy of the summation functions and the Lemma 2 and Lemma 4 we get an equivalence $\mathtt{thm} : \mathtt{Fin}\ (\mathtt{sum}^A\ (\mathtt{f} \circ \pi)) \simeq \mathtt{Fin}\ (\mathtt{sum}^B\ \mathtt{f})$. Since $\mathtt{Fin}$ is injective (i.e $\mathtt{Fin}\ \mathtt{m} \simeq \mathtt{Fin}\ \mathtt{n} \to \mathtt{m} \equiv \mathtt{n}$) the proof is complete. □

$$\begin{tikzcd}
\mathtt{Fin}\ (\mathtt{sum}^A\ (\mathtt{f} \circ \pi)) \arrow[r, leftrightarrow, "\text{thm}"] \arrow[d, leftrightarrow, "\mathtt{sum}^A\ \text{adequate}"'] & \mathtt{Fin}\ (\mathtt{sum}^B\ \mathtt{f}) \arrow[d, leftrightarrow, "\mathtt{sum}^B\ \text{adequate}"] \\
\Sigma\ \mathtt{A}\ (\mathtt{Fin} \circ \mathtt{f} \circ \pi) \arrow[r, leftrightarrow, "\text{lemma 4}"'] & \Sigma\ \mathtt{B}\ (\mathtt{Fin} \circ \mathtt{f})
\end{tikzcd}$$

Lemma 5. *Given two summation functions* sum^A *and* sum^B *for type* A *and type* B, *if both are adequate they satisfy the commutation property that* $\mathit{sum}^A\ (\lambda\ a \to \mathit{sum}^B\ (\lambda\ b \to f\ (a\ ,\ b)))$ *is equal to* $\mathit{sum}^B\ (\lambda\ b \to \mathit{sum}^A\ (\lambda\ a \to f\ (a\ ,\ b)))$.

Proof. By adequacy of $\mathtt{sum}^A$ and $\mathtt{sum}^B$ and the type equivalence between $\Sigma\ \mathtt{A}\ \lambda\ \mathtt{x} \to \Sigma\ \mathtt{B}\ \lambda\ \mathtt{y} \to \mathtt{C}\ \mathtt{x}\ \mathtt{y}$ and $\Sigma\ \mathtt{B}\ \lambda\ \mathtt{y} \to \Sigma\ \mathtt{A}\ \lambda\ \mathtt{x} \to \mathtt{C}\ \mathtt{x}\ \mathtt{y}$. □

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

Theorem 13. *Assume for a type* A *that we have a boolean equality test* $\_{==}\_$ *such that, for all* x *and* y *of type* A, *the type* $(x == y) \equiv 1_2$ *is equivalent to* $x \equiv y$. *Furthermore, assume an adequate summation function* sum, *from which we derive a counting function* count. *Then, for all* x, *the equation* $\mathit{count}\ (\lambda\ y \to x == y) \equiv 1$ *holds.*

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

Lemma 6. *The type family* Fin *is a monoid homomorphism from* $(\mathbb{N}\ ,\ 0\ ,\ \_{+}\_)$ *to* $(\star\ ,\ \mathbb{0}\ ,\ \_{\uplus}\_)$.

Proof. By standard type equivalences, $\mathtt{Fin}\ 0 \simeq \mathbb{0}$ and for all $\mathtt{m}$ and $\mathtt{n}$, then $\mathtt{Fin}\ (\mathtt{m} + \mathtt{n}) \simeq (\mathtt{Fin}\ \mathtt{m} \uplus \mathtt{Fin}\ \mathtt{n})$. □

Theorem 14. *For any type* A *and exploration function* $e^A : \mathit{Explore}\ A$ *such that* $\Sigma^e\ e^A$ *is an adequate* Σ*-type then* $\mathit{sum}\ e^A$ *is an adequate summation function.*

Proof. To give an equivalence $\mathtt{Fin}\ (\mathtt{e}^A\ 0\ \_{+}\_\ \mathtt{f}) \simeq \Sigma\ \mathtt{A}\ (\mathtt{Fin} \circ \mathtt{f})$, instantiate $\mathtt{Adequate\text{-}\Sigma}$ with $\mathtt{F}$ being $\mathtt{Fin} \circ \mathtt{f}$ to get an equivalence $\mathtt{e}^A\ \mathbb{0}\ \_{\uplus}\_\ (\mathtt{Fin} \circ \mathtt{f}) \simeq \Sigma\ \mathtt{A}\ (\mathtt{Fin} \circ \mathtt{f})$. By Theorem 1 and Lemma 6 one get $\mathtt{Fin}\ (\mathtt{e}^A\ 0\ \_{+}\_\ \mathtt{f}) \simeq \mathtt{e}^A\ \mathbb{0}\ \_{\uplus}\_\ (\mathtt{Fin} \circ \mathtt{f})$ and by transitivity the sought after equivalence is reached. □

Lemma 7. *The type family* Fin *is a monoid homomorphism from* $(\mathbb{N}\ ,\ 1\ ,\ \_{*}\_)$ *to* $(\star\ ,\ \mathbb{1}\ ,\ \_{\times}\_)$.

Proof. By standard type equivalences, $\mathtt{Fin}\ 1 \simeq \mathbb{1}$ and for all $\mathtt{m}$ and $\mathtt{n}$, then $\mathtt{Fin}\ (\mathtt{m} * \mathtt{n}) \simeq (\mathtt{Fin}\ \mathtt{m} \times \mathtt{Fin}\ \mathtt{n})$. □

Theorem 15. *For any type* A *and exploration function* $e^A : \mathit{Explore}\ A$ *such that* $\Pi^e\ e^A$ *is an adequate* Π*-type then* $\mathit{product}\ e^A$ *is an adequate product.*

Proof. Proved in a similar way as theorem 14 using Lemma 7. □

5.2 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 part 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 type equivalences. For a probabilistic function which needs to toss a coin, roll a six-sided die and generate a 128-bits key, the type `R` can be any type equivalent to `(𝟚 × D6 × Bits 128)`.

Lemma 8. *Assume an adequate summation function `sum` over a type `R` and let `count` be the derived counting function. Let `f , g : R → 𝟚` such that `count f ≡ count g`, then it is possible to construct an equivalence between `Σ R ( λ x → f x ≡ 1₂ × g x ≡ 0₂ )` and `Σ R ( λ x → f x ≡ 0₂ × g x ≡ 1₂ )`.*

Proof. By proving that `count ( λ x → f x ∧ not ( g x ) ) ≡ count ( λ x → not ( f x ) ∧ g x )` adequacy of `sum` gives the equivalence. The above equality holds since `count f ≡ count ( λ x → f x ∧ g x ) + count ( λ x → f x ∧ not ( g x ) )` and similarly `count g ≡ count ( λ x → f x ∧ g x ) + count ( λ x → not ( f x ) ∧ g x )` therefore since `count f ≡ count g` by assumption we can conclude by canceling `count ( λ x → f x ∧ g x )`. □

Lemma 9. *Given any type `R`, two functions `f , g : R → 𝟚` and an equivalence `e₀ : Σ R ( λ x → f x ≡ 1₂ × g x ≡ 0₂ ) ≃ Σ R ( λ x → f x ≡ 0₂ × g x ≡ 1₂ )`, it is possible to construct an equivalence `e₁ : R ≃ R` such that for all `x`, `f x ≡ g ( e₁ x )` holds.*

Proof. The equivalence `e₁` will be its self inverse. If `f x ≡ g x` it will be the identity, otherwise it will be either `e₀` or `e₀⁻¹` depending on which case we are in. This relies on `e₀ ( e₁ x ) ≡ x` and `e₁ ( e₀ x ) ≡ x` which follows from `e₀` being an equivalence. Furthermore the equivalence have been constructed so that `f x ≡ g ( e₁ x )` holds. □

Theorem 16. *Assume an adequate summation function `sum` over a type `R` and let `count` be the derived counting function. For two events `f , g : R → 𝟚`, such that `f` and `g` have the same probability of occuring i.e `count f ≡ count g`, it is possible to construct an equivalence `π : R ≃ R` such that `f x ≡ g ( π x )`.*

Proof. By Lemma 8 and Lemma 9. □

Corollary 4. *Two events `f g : R → 𝟚` have the same probability of occurring if and only if there is a type equivalence `π : R ≃ R` such that `f` is equal to `g ∘ π`.*

Proof. Combining Theorem 12 and Theorem 16. □

Corollary 5. *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 13. □

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.

Lemma 10. *For any type `A`, and exploration function `eᴬ : Explore A`, two events `f , g : A → 𝟚`, we have `count eᴬ f + count eᴬ g ≡ count eᴬ ( λ x → f x ∧ g x ) + count eᴬ ( λ x → f x ∨ g x )` where `count` is defined in Figure 2.*

Proof. By Theorem 3 we only need to show[9] that for all `x`, `f x + g x ≡ ( f x ∧ g x ) + ( f x ∨ g x )` which is trivial. □

Examples of using type equivalences 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. These symmetries can be exploited by showing the event spaces to be equivalent as types.

Examples from cryptography: Internally an encryption scheme often works using group structures. Assuming an arbitrary group `( G , 0 , _⊕_ , -_ )`, 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 `λ x → x ⊕ m` is an equivalence for some `m` then adding a random value to `m` is indistinguishable from taking a random value. This indstinguishability is proven by showing that, for all observers `O : G → ℕ`, `sum ( λ x → O( x ⊕ m ) )` is equal to `sum ( λ x → O( x ) )`, due to Theorem 12. 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 = ( gʸ , gᶻ ● m )` where both `gʸ` and `gᶻ` can be considered to be random. Hence the adversary will not learn anything about the message `m`.

[9] Here the coercion between `𝟚` and `ℕ` is silent.

6. Discussion

6.1 Related work

Free Theorems Involving Type Constructor Classes: J. Voigtländer[15] shows how to extend HASKELL relational parametricity to constructor classes. In particular one application is to make the use of difference lists transparent. He is defining a `ListLike` type class which is presented differently but equivalent to our three parameters for explorations. His Theorem 6 corresponds to our Corollary 2, his Theorem 7 is similar to our Theorem 1. These two theorems are both fully formalised in development.

The Big_Operators theory in Isabelle: Another development of big operators can be found in Isabelle [11]. 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 functions 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 [3] for COQ has 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 2.2). Another way of defining `reduceBig` would be `reduceBig p l = filter-explore p (foldMap` ^{L} `l)`.

In `bigops` [3], 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 13 states that every adequate exploration satisfies this criterion.

The ALEA library: The COQ library ALEA [1] 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]) →`^{m} `[0,1]`. Here `[0,1]` represents the real numbers between `0.0` and `1.0`, and `_→`^{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 `sum`^{R} `f` holds[10]:

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

[10] We silently coerce `→`^{m} to `→` and `𝟚▷[0,1]` is measuring the likelihood of getting `1`$_{2}$.

6.2 Future work

Beyond `Foldable`: `Traversable` and lenses The type class `Foldable` is hardly the only one with polymorphic methods. For instance the type class `Traversable` has a similar structure. While it has algebraic laws [7], we conjecture they hold by parametricity as well. It would be interesting and challenging to formally carry these results in AGDA.

The lens package [8] is really designed with parametricity in mind. For instance, the library relies on the fact that a monoid is exactly a constant applicative functor, or that a functor which is both covariant and contravariant is necessarily constant. Formalising these constructions should contribute the further design and development of this kind of library.

Parametricity of higher inductive types: We made the choice to go towards homotopy oriented type theory. Moreover we manually implement the parametricity results which avoids the concern on the combinations with univalence [14]. Still we wonder on the interactions. Do higher inductive types [9] enjoy free-theorems in a similar way?

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 3 allows us to distinguish based on the order of elements. To remedy this one can instead use a higher inductive type [9]. The inductive type of binary trees (`Tree`) can be upgraded to a higher inductive type (`FreeCMon`) where the laws for commutative monoids are added as extra equalities. This type `FreeCMon` corresponds to the free commutative monoid. We conjecture that the induction on the type `FreeCMon` corresponds to a refienment of exploration functions where the operator enjoys a commutative monoid structure.

```
data FreeCMon (A : ★) : ★ where
  η       : A → FreeCMon A
  ϵ       : FreeCMon A
  _⊕_     : (l r : FreeCMon A) → FreeCMon A
  comm    : ∀ x y   → x ⊕ y ≡ y ⊕ x
  assoc   : ∀ x y z → (x ⊕ y)⊕ z ≡ x ⊕(y ⊕ z)
  neutral : ∀ x     → ϵ ⊕ x ≡ x
```

6.3 Conclusion

This work presents a way to reason formally about `foldMap`, or as we call them, explorations. No algebraic laws are stated for `Foldable` but some algebraic properties can be recovered by parametricity. We gave a detailed account on how different monoids or monoid homomorphisms interact with explorations. Additionally all the results present in this paper have been fully mechanised in AGDA. Dependent types do not only provide a common framework for programs and proofs but also enable new techniques such as exhaustively exploring a finite type or building a type from an exploration. We showed how type equivalences can establish the adequacy for big operators such as Σ^{e}, Π^{e}, `sum` and `product`. We made this work as a contribution to the safe use of parametricity results in functional programming.

References

[1] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in coq. *Science of Computer Programming*, 74(8):568 – 589, 2009.

[2] J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 345–356, New York, NY, USA, 2010. ACM.

[3] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In O. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer Berlin Heidelberg, 2008.

[4] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.

[5] D. Gustafsson and N. Pouillard. crypto-agda, 2012-2015. `https://github.com/crypto-agda/crypto-agda`.

[6] D. Gustafsson and N. Pouillard. crypto-agda, 2013-2015. `https://github.com/crypto-agda/explore`.

[7] M. Jaskelioff and O. Rypacek. An investigation of the laws of traversals. In *MSFP*, pages 40–49, 2012.

[8] E. A. Kmett. The lens package, 2012-2014. `https://github.com/ekmett/lens`.

[9] P. Lumsdaine and M. Shulman. Higher inductive types. 2013. In preparation.

[10] C. Mcbride. Elimination with a motive. In *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES00), volume 2277 of LNCS*, pages 197–216. Springer-Verlag, 2002.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. Theory big_operators. `http://isabelle.in.tum.de/library/HOL/HOL/Big_Operators.html`.

[12] U. Norell. *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.

[13] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

[14] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Aug. 2013.

[15] J. Voigtländer. Free theorems involving type constructor classes. In A. Tolmach, editor, *14th International Conference on Functional Programming, Edinburgh, Scotland, Proceedings*, volume 44 of *SIGPLAN Notices*, pages 173–184. ACM Press, Sept. 2009.

[16] P. Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, Sept. 1989.

A. Agda development, selected parts

A.1 Listing of Type Equivalences

(★ , (_×_ , 𝟙) , (_⊎_ , 𝟘)) is a commutative semiring up to equivalence.

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

```
Fin-0-𝟘 : Fin 0 ≃ 𝟘
Fin-1-𝟙 : Fin 1 ≃ 𝟙
Fin-2-𝟚 : Fin 2 ≃ 𝟚
Fin-+-⊎ : Fin (m + n) ≃ Fin m ⊎ Fin n
Fin-*-× : Fin (m * n) ≃ Fin m × Fin n
Fin-Σ   : Fin (sum^A f) ≃ Σ A (Fin ∘ f)
Fin-Π   : Fin (prod^A f) ≃ Π A (Fin ∘ f)
```

```
Σ-𝟙   : Σ 𝟙 F ≃ F 0_1
Σ-𝟚   : Σ 𝟚 F ≃ F 0_2 ⊎ F 1_2
Σ-⊎   : Σ (A ⊎ B) F
      ≃ Σ A (F ∘ inl) ⊎ Σ B (F ∘ inr)
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-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 0_1
Π-𝟚   : Π 𝟚 A ≃ A 0_2 × A 1_2
Π-⊎   : Π (A ⊎ B) C ≃ Π A (C ∘ inl) × Π B (C ∘ inr)
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.2 Exploring Σ-types

```
record ⟦Σ⟧
    {A_1 A_2 : ★}
    {B_1 : A_1 → ★}{B_2 : A_2 → ★}
    (A_r : A_1 → A_2 → ★)
    (B_r : (A_r ⟦→⟧ ⟦★⟧) B_1 B_2)
    (p_1 : Σ A_1 B_1) (p_2 : Σ A_2 B_2) : ★ where
  constructor _⟦,⟧_
  field
    ⟦fst⟧ : A_r (fst p_1) (fst p_2)
    ⟦snd⟧ : B_r ⟦fst⟧ (snd p_1) (snd p_2)
```

```
module _
  {A_0 : ★} {A_1 : ★} {A_r : ⟦★⟧ A_0 A_1}
  {B_0 : A_0 → ★} {B_1 : A_1 → ★}
  {B_r : (A_r ⟦→⟧ ⟦★⟧) B_0 B_1}
  {e^A_0 : Explore A_0}
  {e^A_1 : Explore A_1}
  (e^A_r : ⟦Explore⟧ A_r e^A_0 e^A_1)
  {e^B_0 : ∀ x → Explore (B_0 x)}
  {e^B_1 : ∀ x → Explore (B_1 x)}
  (e^B_r : ∀ {x_0 x_1}(x_r : A_r x_0 x_1) →
           ⟦Explore⟧ (B_r x) (e^B_0 x_0) (e^B_1 x_1))
where
  ⟦exploreΣ⟧ : ⟦Explore⟧ (⟦Σ⟧ A_r B_r)
                          (exploreΣ e^A_0 e^B_0)
                          (exploreΣ e^A_1 e^B_1)
  ⟦exploreΣ⟧ M_r ε_r ⊕_r f_r =
     e^A_r M_r ε_r ⊕_r λ x_r →
       e^B_r x_r M_r ε_r ⊕_r λ y_r →
         f_r (x_r ⟦,⟧ y_r)
```

```
module _
  {A : ★}
  {B : A → ★}
  {e^A : Explore A} {e^B : ∀ x → Explore (B x)}
  (e^AP : Explore^P e^A)
  (e^BP : ∀ x → Explore^P (e^B x))
where
  exploreΣ^P : Explore^P (exploreΣ e^A e^B)
  exploreΣ^P e^AP e^BP P ε^P ⊕^P f^P =
    Pe^A (λ e → P (λ _ _ _ → e _ _ _)) ε^P ⊕^P λ x →
      Pe^B x (λ e → P (λ _ _ _ → e _ _ _)) ε^P ⊕^P λ y →
        f^P (x , y)
```