Formalizing Restriction Categories

JAMES CHAPMAN
TARMO UUSTALU
and
NICCOLÒ VELTRI
Institute of Cybernetics, Tallinn University of Technology
Akademia tee 21, 12618 Tallinn, Estonia,
{james,tarmo,niccolo}@cs.ioc.ee

Restriction categories are an abstract axiomatic framework by Cockett and Lack for reasoning about (generalizations of the idea of) partiality of functions. In a restriction category, every map defines an endomap on its domain, the corresponding partial identity map. Restriction categories cover a number of examples of different flavors and are sound and complete with respect to the more synthetic and concrete partial map categories. A partial map category is based on a given category (of total maps) and a map in it is a map from a subobject of the domain.

In this paper, we report on an Agda formalization of the first chapters of the theory of restriction categories, including the challenging completeness result. We explain the mathematics formalized, comment on the design decisions we made for the formalization, and illustrate them at work.

1. INTRODUCTION

Partial functions are used everywhere in mathematics and in programming, but they present a problem for type-theoretical formalization of mathematics and dependently typed programming (DTP). The techniques used for dealing with them tend to be either ad-hoc and ill-justified theoretically or are too involved to be really practical. This paper is motivated by our interest in finding treatments that are both theoretically clean and practical.

Category theory knows several accounts of partial functions on different levels of abstraction, notably Cockett and Lack's restriction categories [7]. Restriction categories are an abstract axiomatic approach to partiality stipulating that every partial function must define a partial endofunction on its domain, the corresponding partial identity, meeting certain equational conditions. Restriction categories cover a number of examples and are sound and complete with respect to partial map categories. The latter are a concrete synthetic model of partiality. A partial map category is based on a given category (of total functions) and defines that a partial function is nothing else than a total function from an acceptable subset of the domain. A restriction category is (isomorphic to) a partial map category if the idempotents corresponding to restrictions split, which intuitively means that

This research was supported by the ERDF funded Estonian CoE project EXCS and ICT National Programme project "Coinduction", the Estonian Ministry of Research and Education target-financed research theme no. 0140007s12 and the Estonian Science Foundation grants no. 9219 and 9475.

the domains of definedness specified as maps by those partial identities that are restrictions must exist in the category also as objects. That can always be achieved by splitting the restriction idempotents of a given restriction category formally.

It is natural to ask whether type-theoretical proof assistants and programming languages could possibly benefit from restriction categories or other category-theoretical approaches to partiality. We present the first results of a study in this direction. We describe a formalization in the DTP language Agda [14] of the first chapters of the theory of restriction categories, in particular the proof of their completeness with respect to partial map categories. For the moment, this is, above all, a formalization effort. We hope that in the future such a development can become the cornerstone of a flexible framework for partiality in DTP languages allowing one to program and reason about partial functions at different levels of abstraction.

The paper is organized as follows. First, in Section 2, we review partial map categories, restriction categories and the completeness theorem in customary mathematical notation. Then, in Section 3, we proceed to describing our Agda formalization. We explain the design decisions we made and the general structure of the formalization and then present the development: definitions of a category; monic map; isomorphism; pullback; a stable system of monics and the partial map category for a given category and stable system of monics; and restriction category and the soundness and completeness theorems.

We used Agda 2.4.2.3 and Agda Standard Library 0.9 for this development. The full Agda code is available online at
`https://github.com/jmchapman/restriction-categories`.

Related work. Restriction categories are a minimalist axiomatic approach to partiality due to Cockett and Lack [7] and generalize the early untyped axiomatization by Menger [13]. In earlier work, Di Paola and Heller [11], and Robinson and Rosolini [15] had similar axiomatics for partial products where one asked for more structure from the start. Often partial maps are the Kleisli maps of a monad on the total map category—this is the situation of partial map classification [3, 8]. Further specializations are categorical axiomatic approaches to recursion, computability and even complexity [11, 6, 10].

In type theory partial functions can be represented in a number of ways. Capretta's computability-theoretically motivated approach [4] is to use Kleisli maps of the so-called delay monad (the constructively viable alternative to the maybe monad). A natural idea is that a partial function is a total function on a subset of the domain given by a predicate. From a general recursive specification of a function, one can read off an inductive definition of such a definedness predicate [1]. Bove, Krauss and Sozeau [2] have given a systematic overview of partiality and recursion in type-theoretic tools.

Agda [14] is a DTP language with a Haskell-inspired syntax; in Agda, proofs are developed exactly as programs, and no difference is made between propositions and sets.

2. THE MATHEMATICS OF PARTIALITY

In this section we present an overview, given in a traditional mathematical style, of the results that we formalized in Agda. All the proofs are given in Section 3.

2.1 Partial Map Categories

Partial map categories are a synthetic approach to partiality. A partial map category is based on some given category whose maps one wants to regard as total.

The idea then is that a partial map is just a total map from a subobject of the domain, the "domain of definedness". It is ok to accept only certain subobjects as domains of definedness. But the collection of acceptable subobjects must satisfy some closure conditions.

Definition 1. A *stable system of monics* for a category $\mathbb{X}$ is a collection $\mathcal{M}$ of monics of $\mathbb{X}$ containing all isomorphisms and closed under composition and arbitrary pullbacks.

(Note that built into this definition is existence of arbitrary pullbacks of monics from $\mathcal{M}$.)

Definition 2. Given a category $\mathbb{X}$ and a stable system of monics $\mathcal{M}$ for it, the corresponding *partial map category* $\mathbf{Par}(\mathbb{X}, \mathcal{M})$ is given as follows:

Objects: objects in $\mathbb{X}$.

Maps: a map from A to B is a span

$$\begin{tikzcd} & A' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & \\ A & & B \end{tikzcd}$$

in $\mathbb{X}$, with $m \in \mathcal{M}$.

Identities: identity on A is the span

$$\begin{tikzcd} & A \arrow[dl, hook', "\mathrm{id}"'] \arrow[dr, "\mathrm{id}"] & \\ A & & A \end{tikzcd}$$

(note that $\mathcal{M}$ contains all isomorphisms, so all identities).

Composition: composition of spans

$$\begin{tikzcd} & A' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & & & B' \arrow[dl, hook', "m'"'] \arrow[dr, "g"] & \\ A & & B & B & & C \end{tikzcd}$$

is given in terms of the pullback of f along m' by

$$\begin{tikzcd} & & W \arrow[dl, hook', "h"'] \arrow[dr, "k"] \arrow[dd, phantom, "\vee" very near start] & & \\ & A' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & & B' \arrow[dl, hook', "m'"'] \arrow[dr, "g"] & \\ A & & B & & C \end{tikzcd}$$

(note that $\mathcal{M}$ is closed under arbitrary pullbacks and composition).

Equality of maps in $\mathbf{Par}(\mathbb{X}, \mathcal{M})$ is defined up to isomorphism of spans: two maps (m, f) and (m', f') between two objects A and B are considered equal, if there exists an isomorphism u such that the triangles in the following diagram commute.

$$\begin{tikzcd}
& A' \arrow[d, "u", "\wr"'] \arrow[ddl, hook', bend right, "m"'] \arrow[ddr, bend left, "f"] & \\
& A'' \arrow[dl, hook', "m'"] \arrow[dr, "f'"'] & \\
A & & B
\end{tikzcd}$$

(As a consequence, it is unproblematic that pullbacks are uniquely determined only up to isomorphism.)

A map (m, f) is called *total*, if m is an isomorphism.

$\mathbb{X}$ is a subcategory of $\mathbf{Par}(\mathbb{X}, \mathcal{M})$.

2.2 Restriction Categories

Restriction categories are an axiomatic formulation of categories of "partial functions". Very little is stipulated: any partial function must define a partial endofunction, intuitively the corresponding partial identity function on the domain, satisfying some equational conditions.

Definition 3. A *restriction category* is a category $\mathbb{X}$ together with an operation called *restriction* that associates to every $f : A \to B$ a map $\overline{f} : A \to A$ such that

R1 $f \circ \overline{f} = f$
R2 $\overline{g} \circ \overline{f} = \overline{f} \circ \overline{g}$ for all $g : A \to C$
R3 $\overline{g} \circ \overline{f} = \overline{g \circ \overline{f}}$ for all $g : A \to C$
R4 $\overline{g} \circ f = f \circ \overline{g \circ f}$ for all $g : B \to C$

The restriction of a map $f : A \to B$ should be thought of as a "partial identity function" on A, a kind of a specification, in the form of a map, of the "domain of definedness" of f.

A map $f : A \to B$ of $\mathbb{X}$ is called *total*, if $\overline{f} = \mathrm{id}_A$. Total maps define a subcategory $\mathbf{Tot}(\mathbb{X})$ of $\mathbb{X}$.

Definition 4. A *restriction functor* between restriction categories $\mathbb{X}$ and $\mathbb{Y}$, with restrictions $\overline{(-)}$ resp. $\widetilde{(-)}$, is a functor F between the underlying categories such that $F\overline{f} = \widetilde{Ff}$.

Restriction categories and restriction functors form a category.

LEMMA 1. *In a restriction category*

(i) monic maps are total, i.e., $\overline{f} = id_A$ *for any monic map* $f : A \to B$*;*
(ii) for any map $f : A \to B$*, its restriction* $\overline{f}$ *is an idempotent, i.e.,* $\overline{f} \circ \overline{f} = \overline{f}$*;*
(iii) the restriction operation itself is idempotent, i.e., $\overline{\overline{f}} = \overline{f}$ *for any map* $f : A \to B$*;*
(iv) $\overline{g \circ f} = \overline{\overline{g} \circ f}$ *for any maps* $f : A \to B$ *and* $g : B \to C$*.*

We compare the pen-and-paper and Agda proofs of Lemma 1 in Section 3.6.

Example 1. The category **Set** of sets and functions (and more generally any category $\mathbb{X}$) is a restriction category with the trivial restriction $\overline{f} = \mathrm{id}$. The category **Par**(**Set**, "all bijections") is isomorphic, as a restriction category, to **Set**.

Example 2. The category **Pfn** of sets and partial functions is a restriction category with the restriction

$$\overline{f}(x) = \begin{cases} x & \text{if } f(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Pfn is the Kleisli category of the "maybe" monad defined by $\mathsf{Maybe}\ A = A + 1$. The category **Par**(**Set**, "all injections") is isomorphic, as a restriction category, to **Pfn**.

Example 3. The subcategory **Prfn** of **Pfn** given by the object $\mathbb{N}$ and all unary partial recursive functions is a restriction category with restriction as defined in Example 2. Note that, for a partially recursive function, its restriction is also partially recursive. No partial map category is isomorphic, as a restriction category, to the restriction category **Prfn**. The reason is that **Prfn** does not have objects for all domains of definition of the maps of **Prfn**, i.e., recursively enumerable sets.

THEOREM 1. *Any partial map category is a restriction category, with the restriction operation given by*

$$\begin{tikzcd} & A' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & \\ A & & B \end{tikzcd} \quad \mapsto \quad \begin{tikzcd} & A' \arrow[dl, hook', "m"'] \arrow[dr, "m"] & \\ A & & A \end{tikzcd}$$

2.3 Idempotents, Splitting Idempotents

Recall that an endomap $e : A \to A$ is called an *idempotent*, if $e \circ e = e$. It is called a *split idempotent*, if there exists an object B and two arrows $s : B \to A$ (*section*) and $r : A \to B$ (*retraction*) such that the following diagrams commute.

$$\begin{tikzcd} A \arrow[d, "e"'] \arrow[dr, "r"] & \\ A & B \arrow[l, hook', "s"] \end{tikzcd} \qquad \begin{tikzcd} & B \arrow[dd, equal] \\ A \arrow[ur, two heads, "r"] & \\ & B \arrow[ul, hook', "s"] \end{tikzcd}$$

(it is automatic that s is monic and r epic).

Every split idempotent is an idempotent. In the converse direction, idempotents do not always split. But one can take any collection $\mathcal{E}$ of idempotents of a category $\mathbb{X}$ that includes all identity maps and formally split them by moving to another category $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ defined by:

Objects: idempotents from $\mathcal{E}$.

Maps: a map from $e : A \to A$ to $e' : B \to B$ is a map $f : A \to B$ of $\mathbb{X}$ such that

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "e"'] & B \\ A \arrow[r, "f"'] & B \arrow[u, "e'"'] \end{tikzcd}$$

Identities: identity on $e : A \to A$ is e.

Composition: inherited from $\mathbb{X}$.

When $\mathcal{E}$ is the collection of all idempotents of $\mathbb{X}$, the category $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ is known as the Karoubi envelope of $\mathbb{X}$.

$\mathbb{X}$ embeds fully in $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ because the collection $\mathcal{E}$ contains all the identities. Moreover, all idempotents from $\mathcal{E}$ split in $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$: given an idempotent $e : A \to A$ from $\mathcal{E}$, the corresponding map $e : \mathrm{id}_A \to \mathrm{id}_A$ in $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ splits via the object $e : A \to A$ with section $e : e \to \mathrm{id}_A$ and retraction $e : \mathrm{id}_A \to e$.

Lemma 2. *Given a restriction category $\mathbb{X}$ and any collection $\mathcal{E}$ of idempotents of $\mathbb{X}$, $\mathbb{X}$ embeds fully, as a restriction category, into $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$, with the restriction of $f : e \to e'$ given by $\widehat{f} = \overline{f} \circ e$.*

In a restriction category $\mathbb{X}$, we call a map $e : A \to A$ a *restriction idempotent*, if $e = \overline{e}$. Lemma 1(ii) tells us that every restriction idempotent is an idempotent. It follows from Lemma 1(iii) that restriction idempotents are precisely those maps e for which $e = \overline{f}$ for some f.

$\mathbb{X}$ is called a *split restriction category*, if all of its restriction idempotents split.

Lemma 3. *Given a restriction category $\mathbb{X}$, for $\mathcal{R}$ the collection of all restriction idempotents, the restriction category $\mathbf{Split}_{\mathcal{R}}(\mathbb{X})$ is a split restriction category.*

The lemma is proved by observing that every restriction idempotent $f : e \to e$ of $\mathbf{Split}_{\mathcal{R}}(\mathbb{X})$ is a restriction idempotent of $\mathbb{X}$ (as $f = \widehat{f} = \overline{f} \circ e = \overline{f} \circ \overline{e} = \overline{\overline{f} \circ e}$) and therefore an object of $\mathbf{Split}_{\mathcal{R}}(\mathbb{X})$. We can therefore split it via f (as an object) with the section $f : f \to e$ and restriction $f : e \to f$.

Example 4. In **Prfn**, restriction idempotents do not split. By splitting the restriction idempotents of **Prfn**, we embed it fully, as a restriction category, into a restriction category $\mathbf{Prfn}^*$, where an object is a recursively enumerable subset of $\mathbb{N}$ and a map between two such sets A and B is a partial recursive function between A and B, by which we mean a partial recursive function $f : \mathbb{N} \to \mathbb{N}$ such that $\mathrm{dom}(f) \subseteq A$ and $\mathrm{rng}(f) \subseteq B$. Its restriction is the corresponding partial identity function on A.

In the subcategory $\mathbf{Tot}(\mathbf{Prfn}^*)$, a map between A and B is a total recursive function between A and B, i.e., a partial recursive function $f : \mathbb{N} \to \mathbb{N}$ such that $\mathrm{dom}(f) = A$ and $\mathrm{rng}(f) \subseteq B$.

2.4 Completeness

If all restriction idempotents of a restriction category split, which intuitively means that all domains of definedness are present in it as objects, then the restriction category is a partial map category on its subcategory of total maps.

Theorem 2. *Every split restriction category $\mathbb{X}$ is isomorphic, as a restriction category, to a partial map category on the subcategory $\mathbf{Tot}(\mathbb{X})$: for $\mathcal{M}$ the stable system of monics given by the sections of the restriction idempotents of $\mathbb{X}$, it holds that*

$$\mathbb{X} \cong \mathbf{Par}(\mathbf{Tot}(\mathbb{X}), \mathcal{M})$$

From Lemmata 2 and 3 and Theorem 2 we obtain the following corollary.

Corollary 1. *A restriction category $\mathbb{X}$ embeds fully, as a restriction category, into a partial map category.*

Example 5. The restriction category $\mathbf{Prfn}^*$ is isomorphic, as a restriction category, to $\mathbf{Par}(\mathbf{Tot}(\mathbf{Prfn}^*)$, "all total recursive injections").

3. FORMALIZATION

We now illustrate how we formalized the mathematics presented in Section 2 in Agda. The full development is available online at
`https://github.com/jmchapman/restriction-categories`.

We have developed our own library of basic utilities: definitions of categories, functors, monics, isomorphisms, sections, idempotents and pullbacks; proofs of various properties about them, e.g., the pasting lemmas for pullbacks. There is currently no standard library for category theory in Agda. The main part of the formalization consists of definitions of restriction categories, partial map categories; examples thereof; proofs of important lemmata; the construction of splitting of idempotents; proofs of the soundness and completeness Theorems 1 and 2.

The formalization consists of 4,000 lines of code (there are two versions, using "type-in-type" resp. universe polymorphism, see the comments below, both are of the same size). The largest part of the code is the completeness theorem (Section 3.10), followed by the definition of the partial map category over a given category and a stable system of monics (Section 3.5) and the soundness theorem (Section 3.7).

3.1 Design Decisions

We represent algebraic-like structures such as categories as dependent records with fields for the data of the structure and fields for the laws. Typically in our formalization record types are opened before their projection functions are utilized. For example, in Section 3.3, in the definition of `Functor`, the field `Hom` from the record type `Cat` is in scope and it takes a category as its first argument. Sometimes we open only one specific record. For example, in Section 3.4, we fix a particular category `X`. In that section, the field `Hom` is in scope and it corresponds to homsets in `X`. We always specify which terms are in context in a section or in a paragraph. Therefore, the reader should not find difficulties in understanding how to interpret fields of records in different situations.

It is common practice in type theory (and Agda) to use setoids to represent the homset when representing categories, so that the laws are given in term of the equivalence relation of the setoid. For this particular formalisation, using setoids would be especially heavy, as we would need setoids of objects as well as homsets: when constructing the category $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$, we take objects to be idempotent maps from the class $\mathcal{E}$ in the underlying category $\mathbb{X}$. We instead use propositional heterogeneous equality (the identity type), which we find much easier to work with. One issue with using Agda's current implementation of propositional equality in this development is that we require function extensionality and existence of quotient types. These principles are valid in extensional type theory and Hofmann has shown that they are a conservative extension of intensional type theory [12]. We make heavy use of uniqueness of equality proofs, which is provable in Agda.

In this paper, we make use of "type-in-type" (i.e., `Set : Set`) instead of Agda's current, in our view, excessively verbose implementation of universe polymorphism.

Our full Agda formalization comes in two versions: one using "type-in-type", one using universe polymorphism.

3.2 Quotients

Equality of maps in partial map category is defined up to isomorphism of spans. That means that in order to properly formalize partial map categories we need quotients. Agda does not currently support quotients, so we postulate their existence as we do for extensionality. Our implementation of quotients is inspired by Martin Hofmann's inductive-like quotient types [12]. We define a record type `Quotient` for a set `A` and an equivalence relation `R` on `A` using the standard library machinery for equivalence relations. An equivalence relation on a type `A` is a binary relation on `A` together with a proof of it being reflexive, symmetric and transitive (this predicate is called `isEquivalence` in Agda's standard library).

```
EqR : Set → Set
EqR A = Σ (A → A → Set) IsEquivalence
```

The `Quotient` record type has a field `Q` for the set of equivalence classes of `A` and a field `abs` for the canonical projection map `A → Q`. As well as `abs` we have a dependent eliminator `lift`, which lifts (dependent) functions from `A` to functions from `Q`. This operation can only lift `compat`ible functions and hence it takes a compatibility proof as an extra argument. `compat` is a predicate on functions from `A` stating that the function takes related arguments in `A` to equal results. Notice that `abs` is compatible by `sound`. Axiom `liftbeta` states that applying a lifted function to an abstracted argument is the same as applying the function to the argument directly.

```
record Quotient (A : Set)(R : EqR A) : Set where
  open Σ R renaming (proj₁ to _∼_)
  field Q   : Set
        abs : A → Q

  compat : (B : Q → Set)(f : (a : A) → B (abs a)) → Set
  compat B f = ∀{a b} → a ∼ b → f a ≅ f b

  field sound    : compat _ abs
        lift     : (B : Q → Set)(f : (a : A) → B (abs a))
                   (p : compat B f) → (x : Q) → B x
        liftbeta : (B : Q → Set)(f : (a : A) → B (abs a))
                   (p : compat B f)(a : A) →
                   lift B f p (abs a) ≅ f a
```

It is useful to have a version of `compat`, `lift` and `liftbeta` for two-argument functions. Let `A` and `A'` be types and `R` and `R'` equivalence relations on `A` and `A'`. Let `_∼_` and `_∼'_` be the binary relations associated with `R` and `R'` respectively. We fix a quotient of `A` by `R` and a quotient of `A'` by `R'`. The fields of the second quotient are marked with an apostrophe.

```
compat₂ : (B : Q → Q' → Set)
          (f : (a : A)(a' : A') → B (abs a) (abs' a')) → Set
```

```
compat₂ B f = ∀{a b a' b'} → a ∼ a' → b ∼' b' → f a b ≅ f a' b'

lift₂ : (B : Q → Q' → Set)
        (f : (a : A)(a' : A') → B (abs a) (abs' a'))
        (p : compat₂ B f)(x : Q)(x' : Q') → B x x'
lift₂ f p x x' = ?

liftbeta₂ : (B : Q → Q' → Set)
            (f : (a : A)(a' : A') → B (abs a) (abs' a'))
            (p : compat₂ B f)(a : A)(a' : A') →
            lift₂ B f p (abs a) (abs' a') ≅ f a a'
liftbeta₂ = ?
```

In Agda, unfinished parts of a definition are denoted by a question mark `?`. In this paper, we leave some definitions incomplete. We omit some definitions due to reasons of space and/or readability. The full formalization contains no unfinished parts.

Every set together with an equivalence relation on it gives rise to a quotient. This is what we need to postulate in Agda. The record type `Quotient` gives a specification of a quotient, `quot` assumes that this specification holds (is inhabited) for any set and equivalence relation on it.

```
postulate
  quot : (A : Set)(R : EqR A) → Quotient A R
```

3.3 Categories

Categories are described as a record type with fields for the set of objects, the set of maps between two objects, for any object an identity map and for any pair of suitable maps their composition. Further to this, we have three fields for the laws of a category given as propositional equalities between maps.

```
record Cat : Set where
  field Obj  : Set
        Hom  : Obj → Obj → Set
        iden : ∀{A} → Hom A A
        comp : ∀{A B C} → Hom B C → Hom A B → Hom A C
        idl  : ∀{A B}{f : Hom A B} → comp iden f ≅ f
        idr  : ∀{A B}{f : Hom A B} → comp f iden ≅ f
        ass  : ∀{A B C D}{f : Hom C D}{g : Hom B C}{h : Hom A B} →
               comp (comp f g) h ≅ comp f (comp g h)
```

Functors are also described as a record type with fields for the mapping of objects, the mapping of morphisms and the two laws stating that the latter must preserve identities and composition.

```
record Fun (X Y : Cat) : Set where
  field OMap  : Obj X → Obj Y
        HMap  : ∀{A B} → Hom X A B → Hom Y (OMap A) (OMap B)
        fid   : ∀{A} → HMap (iden X {A}) ≅ iden Y {OMap A}
        fcomp : ∀{A B C}{f : Hom X B C}{g : Hom X A B} →
```

```
HMap (comp X f g) ≅ comp Y (HMap f) (HMap g)
```

The identity functor has identity maps as mapping of objects and mapping of morphisms, and reflexivity proves the functor laws.

```
idFun : {X : Cat} → Fun X X
idFun = record{
  OMap = id;
  HMap = id;
  fid = refl;
  fcomp = refl}
```

The properties of functors being full and faithful are given as predicates on functors.

```
Full : {X Y : Cat}(F : Fun X Y) → Set
Full {X}{Y} F =
  ∀{A B}{f : Hom Y (OMap F A) (OMap F B)} →
    Σ (Hom X A B) λ g → HMap F g ≅ f

Faithful : {X Y : Cat}(F : Fun X Y) → Set
Faithful {X} F =
  ∀{A B}{f g : Hom X A B} → HMap F f ≅ HMap F g → f ≅ g
```

3.4 Monics, Isomorphisms and Pullbacks

In this section, we work in a particular category `X`. In Agda, this corresponds to working in a module parameterized by a category `X`. Moreover, as already discussed in Section 3.1, we open the specific record `X`. This implies that, for example, the projections `Obj`, `Hom` and `iden` refer to objects, homsets and identity morphisms in the category `X`.

3.4.1 *Monic Maps.* A map `f` is monic, if, for any suitable maps `g` and `h`, we have `comp f g ≅ comp f h` implies `g ≅ h`.

```
Mono : ∀{A B}(f : Hom A B) → Set
Mono f = ∀{C}{g h : Hom C _} → comp f g ≅ comp f h → g ≅ h
```

We prove a lemma `idMono` stating that every identity map is monic. An equational proof starts with the word `proof` and ends with the symbol ■. The proof is an alternating sequence of expressions and justifications. It is very close to how one would write it on paper, but we do not gloss over minor details such as appeals to associativity of composition in a category. We must also be very precise about where in an expression we apply a rewrite rule.

```
idMono : ∀{A} → Mono (iden {A})
idMono {g = g}{h} p =
  proof
  g
  ≅⟨ sym idl ⟩
  comp iden g
  ≅⟨ p ⟩
  comp iden h
```

```
  ≅⟨ idl ⟩
  h
  ■
```

3.4.2 *Isomorphisms.* Isomorphism is defined as a predicate on maps that is witnessed by a suitable inverse map and proofs of the two isomorphism properties.

```
record Iso {A B : Obj}(f : Hom A B) : Set where
  field inv  : Hom B A
        rinv : comp f inv ≅ iden {B}
        linv : comp inv f ≅ iden {A}
```

We prove that any identity map is trivially an isomorphism, the proof arguments are given by the left identity property `idl` (right identity `idr` also works) of the category `X`.

```
idIso : ∀{A} → Iso (iden {A})
idIso = record{
  inv = iden;
  rinv = idl;
  linv = idl}
```

3.4.3 *Pullbacks.* The definition of pullback is divided into three parts. First we give the definition of a square over a cospan, i.e., a pair of maps `f : Hom A C` and `g : Hom B C` with the same target object. It is a record consisting of an object `W`, two maps `h` and `k` completing the square, and a proof `scom` that the square commutes.

```
record Square {A B C}(f : Hom A C)(g : Hom B C) : Set where
  field W    : Obj
        h    : Hom W A
        k    : Hom W B
        scom : comp f h ≅ comp g k
```

$$\begin{tikzcd} & W \arrow[dl, "h"'] \arrow[dr, "k"] & \\ A \arrow[dr, "f"'] & & B \arrow[dl, "g"] \\ & C & \end{tikzcd}$$

Then we define a map between two squares, called a `SqMap`. It consists of a map `sqMor` between the respective `W` objects of the squares together with proofs of commutation of the two triangles that the map `sqMor` generates.

```
record SqMap {A B C : Obj}{f : Hom A C}{g : Hom B C}
             (sq' sq : Square f g) : Set where
  field sqMor   : Hom (W sq') (W sq)
        leftTr  : comp (h sq) sqMor ≅ h sq'
        rightTr : comp (k sq) sqMor ≅ k sq'
```

$$\begin{tikzcd}
 & W' \arrow[d] \arrow[ddl, bend right, "h'"'] \arrow[ddr, bend left, "k'"] & \\
 & W \arrow[dl, "h"] \arrow[dr, "k"'] & \\
A \arrow[dr, "f"'] & & B \arrow[dl, "g"] \\
 & C &
\end{tikzcd}$$

A pullback of maps `f` and `g` consists of a square `sq` and a universal property `uniqPul`: for any other square `sq'` over `f` and `g`, there exists a unique map between `sq` and `sq'`.

```
record Pullback {A B C}(f : Hom A C)(g : Hom B C) : Set where
  field sq      : Square f g
        uniqPul : (sq' : Square f g) →
                  Σ (SqMap sq' sq) λ u →
                    (u' : SqMap sq' sq) → sqMor u ≅ sqMor u'
```

Later we will need two results regarding pullbacks. The first is the definition of a pullback of a map `f` along the identity map. The other two sides completing the pullback square are `f` and the identity map.

```
trivialSquare : ∀{A B}(f : Hom A B) → Square f iden
trivialSquare {A} f = record{
  W = A;
  h = iden;
  k = f;
  scom =
    proof
    comp f iden
    ≅⟨ idr ⟩
    f
    ≅⟨ sym idl ⟩
    comp iden f
    ■}
```

$$\begin{tikzcd}
 & A \arrow[dl, equal] \arrow[dr, "f"] & \\
A \arrow[dr, "f"'] & & B \arrow[dl, equal] \\
 & B &
\end{tikzcd}$$

Let `sq` be another square over f and the identity map. We call `W` the object, `h` and `k` the two maps that complete the square `sq`, and `scom` the proof that the square commutes. Then `h` together with the straightforward proofs that the two triangles it generates commute is a map between the two squares.

```
trivialSqMap : ∀{A B}(f : Hom A B)(sq : Square f iden) →
```

```
               SqMap sq (trivialSquare f)
trivialSqMap f sq = record{
  sqMor = h sq;
  leftTr = idl;
  rightTr =
    proof
    comp f (h sq)
    ≅⟨ scom sq ⟩
    comp iden (k sq)
    ≅⟨ idl ⟩
    k sq
    ■}
```

$$\begin{tikzcd}
 & W \arrow[d, "h"] \arrow[ddl, bend right, "h"'] \arrow[ddrr, bend left, "k"] & \\
 & A \arrow[dl, equal] \arrow[dr, "f"] & \\
A \arrow[dr, "f"'] & & B \arrow[dl, equal] \\
 & B &
\end{tikzcd}$$

To complete the construction of the pullback, it remains to supply a proof that the map `trivialSqMap f sq` between the arbitary square `sq` and `trivialSquare f` is unique.

```
trivialPullback : ∀{A B}(f : Hom A B) → Pullback f iden
trivialPullback f = record{
  sq = trivialSquare f;
  uniqPul = λ sq →
    trivialSqMap f sq ,
    λ u →
      proof
      h sq
      ≅⟨ sym (leftTr u) ⟩
      comp iden (sqMor u)
      ≅⟨ idl ⟩
      sqMor u
      ■}
```

The second result regarding pullbacks we need is a theorem stating that any two pullbacks over the same maps are isomorphic. The isomorphism is the unique map between the two squares provided by the universal property of pullbacks.

```
pullbackIso : ∀{A B C}{f : Hom A C}{g : Hom B C}
              (p p' : Pullback f g) →
              Iso (sqMor (proj1 (uniqPul p (sq p'))))
pullbackIso p p' = ?
```

$$
\begin{tikzcd}
& W' \arrow[d] \arrow[ddl, bend right, "h'"'] \arrow[ddr, bend left, "k'"] & \\
& W \arrow[dl, "h"] \arrow[dr, "k"'] \arrow[dd, phantom, very near start, "\lrcorner", rotate=-45] & \\
A \arrow[dr, "f"'] & & B \arrow[dl, "g"] \\
& C &
\end{tikzcd}
$$

3.5 Partial Map Categories

Let `X` be a category. A stable system of monics in `X` is a set of maps given by a membership predicate `∈sys` satisfying four properties: every element is monic; all isomorphisms are elements; the set is closed under composition; and the set is closed under pullback along arbitrary maps.

```
record StableSys : Set where
  field ∈sys     : ∀{A B}(f : Hom A B) → Set
        mono∈sys : ∀{A B}{f : Hom A B} → ∈sys f → Mono f
        iso∈sys  : ∀{A B}{f : Hom A B} → Iso f → ∈sys f
        comp∈sys : ∀{A B C}{f : Hom A B}{g : Hom B C} → ∈sys f →
                   ∈sys g → ∈sys (comp g f)
        pul∈sys  : ∀{A B C}(f : Hom A C){m : Hom B C} → ∈sys m →
                   Σ (Pullback f m) λ p → ∈sys (h (sq p))
```

A partial map category is a category defined on a stable system of monics `M` on `X`. The objects are the objects of `X` and the maps are spans which are defined as a record type indexed by source and target objects `A` and `B` consisting of a third object `A'`, two maps `mhom` and `fhom` for the left and right leg of the span, and a proof that the left leg `mhom` is a member of the stable system of monics.

```
record Span (A B : Obj) : Set where
  field A'    : Obj
        mhom  : Hom A' A
        fhom  : Hom A' B
        m∈sys : ∈sys mhom
```

$$
\begin{tikzcd}
& A' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & \\
A & & B
\end{tikzcd}
$$

Equality on spans is defined up to isomorphism. We prove the properties of spans up to this isomorphism, and then use a quotient to work with this isomorphism in the place of equality. Two spans `mf` and `ng` are 'equal', if, for some isomorphism between their source objects, the two triangles it generates commute.

```
record _∼Span∼_ {A B}(mf ng : Span A B) : Set where
  field s        : Hom (A' mf) (A' ng)
        sIso     : Iso s
```

```
leftTr∼  : comp (mhom ng) s ≅ mhom mf
rightTr∼ : comp (fhom ng) s ≅ fhom mf
```

$$
\begin{tikzcd}
 & A' \arrow[dl, hook', bend right, "m"'] \arrow[d, "s", "\wr"'] \arrow[dr, bend left, "f"] & \\
 & A'' \arrow[dl, hook', "n"] \arrow[dr, "g"'] & \\
A & & B
\end{tikzcd}
$$

Notice that, for all `mf ng : Span A B`, the type `mf ∼Span∼ ng` is a proposition, i.e., any two inhabitants of this type are equal. In fact, suppose there are two isomorphisms `s` and `t` between the spans `mf` and `ng`.

$$
\begin{tikzcd}
 & A' \arrow[dl, hook', bend right, "m"'] \arrow[d, shift right, "t"'] \arrow[d, shift left, "s"] \arrow[dr, bend left, "f"] & \\
 & A'' \arrow[dl, hook', "n"] \arrow[dr, "g"'] & \\
A & & B
\end{tikzcd}
$$

In particular, `comp n t ≅ m ≅ comp n s`. Since `n` is monic, we have `s ≅ t`. The relation `_∼Span∼_` forms an equivalence relation.

```
Span∼EqR : ∀{A B} → EqR (Span A B)
Span∼EqR = _∼Span∼_ , ?
```

We quotient `Span A B` by this equivalence relation and we call the result `qspan A B`.

```
qspan : ∀ A B → Quotient (Span A B) Span∼EqR
qspan A B = quot (Span A B) Span∼EqR
```

The carriers `QSpan A B` of such quotients are the homsets in the partial map category.

```
QSpan : ∀ A B → Set
QSpan A B = Quotient.Q (qspan A B)
```

We define shorthand names for referring to the quotient machinery for an arbitrary span, e.g.

```
abs : {A B : Obj} → Span A B → QSpan A B
abs {A}{B} = Quotient.abs (qspan A B)
```

Shorthands for the other fields are given in a similar way. From now on the names `compat`, `sound`, `lift` and `liftbeta` always refer to the respective fields in `qspan A B`. The same applies for the two-argument variants of `compat`, `lift` and `liftbeta`.

The partial map category has sets as objects and `QSpan`s as homsets. Just as homsets are defined in two steps (first `Span`, then `QSpan`), the operations and laws are also defined in two steps. We first define operations on `Span`s and then port them to `QSpan`s. Analogously, we prove the laws up to `_∼Span∼_` and then port them to equality proofs. Note that the whole construction of the partial map category and the soundness proof is performed first up to `_∼Span∼_`. This means that this part of our formalisation could be reused even if one wanted to take the "setoid approach" to formalising category theory in type theory.

The identity span is trivial to describe. The left and right legs are identities and identities are isomorphisms, hence they are available in any stable system of monics.

```
idSpan : {A : Obj} → Span A A
idSpan {A} = record{
  A' = A;
  mhom = iden;
  fhom = iden;
  m∈sys = iso∈sys idIso}
```

The identity maps in the partial map category are given by `abs idSpan`.

Let `ng : Span B C` and `mf : Span A B` be two spans. Let `n` and `m` be the left legs of `ng` and `mf` respectively, `g` and `f` be the right legs, and `n∈` and `m∈` be the proofs that `n` and `m` are in the stable system of monics. In order to form the composite span `compSpan ng mf`, we take the pullback of `f` along `n`. `W` is the object, and `h` and `k` are the two maps that complete the square underlying such pullback. The composite span is given by composing `h` with `m` and `k` with `g`. The span is well defined, since both `h` and `m` are in the stable system of monics, which is closed under composition.

```
compSpan : ∀{A B C} → Span B C → Span A B → Span A C
compSpan ng mf =
  let sq' = sq (proj1 (pul∈sys (fhom mf) (m∈sys ng)))
  in record{
    A' = W sq';
    mhom = comp (mhom mf) (h sq');
    fhom = comp (fhom ng) (k sq');
    m∈sys =
      comp∈sys (proj2 (pul∈sys (fhom mf) (m∈sys ng))) (m∈sys mf)}
```

$$
\begin{tikzcd}
& & W \arrow[dl, hook', "h"'] \arrow[dr, "k"] \arrow[dd, phantom, "\lrcorner", very near start] & & \\
& A' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & & B' \arrow[dl, hook', "n"'] \arrow[dr, "g"] & \\
A & & B & & C
\end{tikzcd}
$$

We need a lemma `∼cong` stating that `_∼Span∼_` is a congruence with respect to composition of spans. `compSpan` is an operation on spans, so when reasoning about spans up to equality, we need that `compSpan` respects it.

```
∼cong : ∀{A B C}{ng n'g' : Span B C}{mf m'f' : Span A B} →
        mf ∼Span∼ m'f' → ng ∼Span∼ n'g' →
        compSpan ng mf ∼Span∼ compSpan n'g' m'f'
∼cong p r = ?
```

Composition is obtained by lifting the function `λ x y → abs (compSpan x y)`, which is compatible with `_∼Span∼_`.

```
qcompSpan : ∀{A B C} → QSpan B C → QSpan A B → QSpan A C
qcompSpan =
  lift₂ _ (λ x y → abs (compSpan x y)) (λ p q → sound (∼cong p q))
```

The function `qcompSpan` is propositionally equal to `abs (compSpan ng mf)`, when applied to terms `abs ng` and `abs mf`.

```
liftbetaComp : ∀{A B C}{ng : Span B C}{mf : Span A B} →
               qcompSpan (abs ng) (abs mf) ≅ abs (compSpan ng mf)
liftbetaComp =
  liftbeta₂ _ (λ x y → abs (compSpan x y))
    (λ p q → sound (∼cong p q)) _ _
```

Next we present the proof of the left identity law for a partial map category. We have to prove that a span composed with the identity span is 'equal' to itself (up to `_∼Span∼_`). Let `mf : Span A B` be a span. Let `A'` be the object, `m` and `f` the left and right legs of `mf`. We take the pullback of the identity map along `f` given by the fact that the identity map is in every stable system of monics. We call `W` the object, and `h` and `k` the two maps that complete the square underlying such pullback. Let `scom` be the proof that the square commutes.

$$
\begin{tikzcd}
 & & W \arrow[dl, hook', "h"'] \arrow[dr, "k"] \arrow[dd, phantom, "\lrcorner" rotate=-45, very near start] & & \\
 & A' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & & B \arrow[dl, equal] \arrow[dr, equal] & \\
A & & B & & B
\end{tikzcd}
$$

We have to find an isomorphism between `W` and `A'` that makes the two generated diagrams commute.

$$
\begin{tikzcd}
 & & W \arrow[dl, hook', "h"'] \arrow[d, "\wr"] \arrow[dr, "k"] & & \\
 & A' \arrow[dl, hook', "m"'] & A' \arrow[dll, hook', "m"] \arrow[drr, "f"'] & B \arrow[dr, equal] & \\
A & & & & B
\end{tikzcd}
$$

Note that there is also another pullback of the identity map along `f`, namely `trivialPullback f`. By the definition of `trivialPullback f`, the underlying morphism of the unique map to it from the given pullback is `h`; by `pullbackiso`, it is an isomorphism We supply `h` and the isomorphism proof together with the proofs that the two triangles in the diagram above commute. The first one is just `refl` and the second one follows immediately from the proof `scom` of the pullback of the identity map along `f`.

$$
\begin{tikzcd}
 & W \arrow[ddl, bend right, "h"'] \arrow[d, "h"] \arrow[ddr, bend left, "k"] & \\
 & A' \arrow[dl, equal] \arrow[dr, "f"] \arrow[dd, phantom, "\lrcorner" rotate=-45, very near start] & \\
A' \arrow[dr, "f"'] & & B \arrow[dl, equal] \\
 & B &
\end{tikzcd}
$$

```
idlSpan : ∀{A B}{mf : Span A B} → compSpan idSpan mf ∼Span∼ mf
idlSpan {mf = mf} = record{
  let p = proj₁ (pul∈sys (fhom mf) (iso∈sys idIso))
      sq' = sq p
  in record{
  s = h sq';
  sIso = pullbackIso (trivialPullback (fhom mf)) p;
  leftTr∼ = refl;
  rightTr∼ = scom sq'}
```

We will use the quotient machinery in conjunction with this proof in the definition of the partial map category.

The composition `qcompSpan` is defined using `lift₂`, therefore, when applied to arguments `abs ng` and `abs mf`, it is propositionally equal to `abs (compSpan ng mf)`. Using this result, we prove the left identity law for the partial map category.

```
qidlSpan : ∀{A B}{x : QSpan A B} → qcompSpan (abs idSpan) x ≅ x
qidlSpan {x = x} =
  lift (λ mf →
          proof
          qcompSpan (abs idSpan) (abs mf)
          ≅⟨ liftbetaComp ⟩
          abs (compSpan idSpan mf)
          ≅⟨ sound idlSpan ⟩
          abs mf
          ■)
        (fixtypes ∘ sound)
        x
```

In the above proof, we used the lemma `fixtypes`, useful to deal with the common situation where we have proofs of two equations with equal right hand sides.

```
fixtypes : {A B C D : Set}{a : A}{b : B}{c : C}{d : D}
           {p : a ≅ b}{q : c ≅ d} → b ≅ d → p ≅ q
fixtypes {p = refl}{refl} refl = refl
```

The right identity law and associativity of `qcompSpan` are proved similarly to the left identity law.

```
Par : Cat
Par = record{
  Obj  = Obj;
  Hom  = QSpan;
  iden = abs idSpan;
  comp = qcompSpan;
  idl  = qidlSpan;
  idr  = ?;
  ass  = ?}
```

3.6 Restriction Categories

A restriction category is a category with a restriction operation, i.e., every map comes with an endomap on its domain subject to four laws.

```
record RestCat : Set where
  field cat  : Cat
        rest : ∀{A B} → Hom cat A B → Hom cat A A
        R1   : ∀{A B}{f : Hom cat A B} → comp cat f (rest f) ≅ f
        R2   : ∀{A B C}{f : Hom cat A B}{g : Hom cat A C} →
               comp cat (rest f) (rest g) ≅
               comp cat (rest g) (rest f)
        R3   : ∀{A B C}{f : Hom cat A B}{g : Hom cat A C} →
               comp cat (rest g) (rest f) ≅
               rest (comp cat g (rest f))
        R4   : ∀{A B C}{f : Hom cat A B}{g : Hom cat B C} →
               comp cat (rest g) f ≅
               comp cat f (rest (comp cat g f))
```

A restriction functor between two restriction categories is a functor between the underlying categories preserving the restriction operation.

```
record RestFun (C D : RestCat) : Set where
  field fun   : Fun (cat C) (cat D)
        frest : ∀{A B}{f : Hom (cat C) A B} →
                rest D (HMap fun f) ≅ HMap fun (rest C f)
```

The identity functor is always a restriction functor.

```
idRestFun : {C : RestCat} → RestFun C C
idRestFun = record{
  fun = idFun;
  frest = refl}
```

We fix a restriction category `X` with underlying category `Xcat`. We prove lemmata `lem1`, `lem2`, `lem3` and `lem4` (Lemma 1 (*i*)-(*iv*) in Section 2.2). We show them here, since we are going to use them later on. Moreover, they are nice examples of the kind of equational reasoning one can do with restriction categories.

```
lem1 : ∀{A B}{f : Hom A B} → Mono f → rest f ≅ iden
lem1 {f = f} p =
  p (proof
```

```
    comp f (rest f)
    ≅⟨ R1 ⟩
    f
    ≅⟨ sym idr ⟩
    comp f iden
    ■)

lem2 : ∀{A B}{f : Hom A B} → comp (rest f) (rest f) ≅ rest f
lem2 {f = f} = proof
  comp (rest f) (rest f)
  ≅⟨ R3 ⟩
  rest (comp f (rest f))
  ≅⟨ cong rest R1 ⟩
  rest f
  ■

lem3 : ∀{A B}{f : Hom A B} → rest (rest f) ≅ rest f
lem3 {f = f} = proof
  rest (rest f)
  ≅⟨ cong rest (sym idl) ⟩
  rest (comp iden (rest f))
  ≅⟨ sym R3 ⟩
  comp (rest iden) (rest f)
  ≅⟨ cong (λ g → comp g (rest f)) (lem1 idMono) ⟩
  comp iden (rest f)
  ≅⟨ idl ⟩
  rest f
  ■

lem4 : ∀{A B C}{f : Hom A B}{g : Hom B C} →
        rest (comp g f) ≅ rest (comp (rest g) f)
lem4 {f = f}{g} = proof
  rest (comp g f)
  ≅⟨ cong (λ f' → rest (comp g f')) (sym R1) ⟩
  rest (comp g (comp f (rest f)))
  ≅⟨ cong rest (sym ass) ⟩
  rest (comp (comp g f) (rest f))
  ≅⟨ sym R3 ⟩
  comp (rest (comp g f)) (rest f)
  ≅⟨ R2 ⟩
  comp (rest f) (rest (comp g f))
  ≅⟨ R3 ⟩
  rest (comp f (rest (comp g f)))
  ≅⟨ cong rest (sym R4) ⟩
  rest (comp (rest g) f)
  ■
```

Notice how equational proofs in Agda look literally like those one would write by hand. E.g., compare the formal proof of `lem4` given above with the following pen-and-paper proof:

$$\overline{g \circ f} = \overline{g \circ (f \circ \overline{f})} = \overline{(g \circ f) \circ \overline{f}} = \overline{g \circ f} \circ \overline{f} = \overline{f} \circ \overline{g \circ f} = \overline{f \circ \overline{g \circ f}} = \overline{\overline{g} \circ f}$$

Restriction categories allow us to work with partial maps in a total setting. However, we still need to be able to identify total maps. In a restriction category, a total map is a map whose restriction is the identity map.

```
record Tot (A B : Obj) : Set where
  field hom     : Hom A B
        totProp : rest hom ≅ iden {A}
```

We need a lemma `totEq` stating that two total maps are equal, if their underlying morphisms are equal. This is a consequence of uniqueness of identity proofs.

```
totEq : ∀{A B}{f g : Tot A B} → hom f ≅ hom g → f ≅ g
totEq p = ?
```

The category `Total` of total maps in `X` inherits its identity `idTot` and composition `compTot` from the underlying category `Xcat`, but we must prove that the totality property `totProp` is satisfied. For the identity map, `idTot` the condition follows from the fact that identity maps are monic `idMono` and monic maps are total `lem1`.

```
idTot : ∀{A} → Tot A A
idTot = record{
  hom = iden;
  totProp = lem1 idMono}
```

Given two total maps `g` and `f`, the totality condition `compTotProp` for the composite `compTot g f` follows from totality of `g` and `f` and `lem4`.

```
compTotProp : ∀{A B C}{g : Tot B C}{f : Tot A B} →
              rest (comp (hom g) (hom f)) ≅ iden
compTotProp {g = g}{f} =
  proof
  rest (comp (hom g) (hom f))
  ≅⟨ lem4 ⟩
  rest (comp (rest (hom g)) (hom f))
  ≅⟨ cong (λ h → rest (comp h (hom f))) (totProp g) ⟩
  rest (comp iden (hom f))
  ≅⟨ cong rest idl ⟩
  rest (hom f)
  ≅⟨ totProp f ⟩
  iden
  ■

compTot : ∀{A B C}(g : Tot B C)(f : Tot A B) → Tot A C
compTot g f = record{
  hom = comp (hom g) (hom f);
  totProp = compTotProp}
```

Having defined identities and composition, we can now define the category of total maps. The `totEq` lemma reduces the laws of a category to those of the underlying category.

```
Total : Cat
Total = record{
  Obj  = Obj;
  Hom  = Tot;
  iden = idTot;
  comp = compTot;
  idl  = totEq idl;
  idr  = totEq idr;
  ass  = totEq ass}
```

3.7 Soundness

The next step in the formalization is the soundness theorem, which states that any partial map category is a restriction category. In order to prove it, we equip the given partial map category with a restriction category structure (a restriction operator, proofs of R1, R2, R3 and R4). We perform the construction in two steps: first on spans and then we port it to quotiented spans, as we did in the definition of partial map categories. We fix a category `X` and a stable system of monics `M`. The restriction on spans simply copies the left leg of a span into the right leg position.

```
restSpan : ∀{A B} → Span A B → Span A A
restSpan mf = record{
  A’ = A’ mf;
  mhom = mhom mf;
  fhom = mhom mf;
  m∈sys = m∈sys mf}
```

We require that restriction respects the equivalence relation on spans. This is easy: the left commuting triangle is copied to the right.

```
∼congRestSpan : ∀{A B}{mf m’f’ : Span A B} → mf ∼Span∼ m’f’ →
                restSpan mf ∼Span∼ restSpan m’f’
∼congRestSpan eq = record{
  s = s eq;
  sIso = sIso eq;
  leftTr∼ = leftTr∼ eq;
  rightTr∼ = leftTr∼ eq}
```

We port the restriction operator on spans to quotiented spans. We first postcompose `restSpan` with `abs`, obtaining a map from `Span A B` to `QSpan A A`. Then we lift this map. Compatibility follows from axiom `sound` and the above proved congruence `∼congRestSpan`.

```
qrestSpan : ∀{A B} → QSpan A B → QSpan A A
qrestSpan = lift (abs ∘ restSpan) (sound ∘ ∼congRestSpan)
```

The function `qrestSpan` is propositionally equal to `abs (restSpan mf)`, when applied to a term `abs mf`.

```
liftbetaRest : ∀{A B}{mf : Span A B} →
               qrestSpan (abs mf) ≅ abs (restSpan mf)
liftbetaRest =
  liftbeta _ (abs ∘ restSpan) (sound ∘ ∼congRestSpan) _
```

To prove R1 for the partial map category, we will use the basic fact that one can construct the following pullback from any monic map.

$$
\begin{tikzcd}
 & A' \arrow[dl, equal] \arrow[dr, equal] \arrow[dd, phantom, "\lrcorner" rotate=-45, very near start] & \\
A' \arrow[dr, hook, "m"'] & & A' \arrow[dl, hook', "m"] \\
 & A &
\end{tikzcd}
$$

```
monicPullback : ∀{A' A}{m : Hom A' A} → Mono m → Pullback m m
monicPullback p = ?
```

R1 states that composing a map `f` with its restriction is the same as `f`. We prove this property up to equivalence of spans first. Let `mf : Span A B` be a span. Let `A'` be the object, `m` and `f` the left and right legs of `mf`, and `m∈` the proof that `m` is in the stable system of monics. Note that there are two pullbacks of `m` along itself: (i) the pullback `monicPullback (mono∈sys m∈)`, where `mono∈sys m∈` is a proof that `m` is monic; (ii) the pullback given by the fact that `m` is in the stable system of monics, and therefore the pullback of `m` along any map exists. We call `W` the object, `h` and `k` the two maps that complete the square underlying the pullback (ii), and `scom` the proof that the square commutes.

$$
\begin{tikzcd}
 & W \arrow[ddl, bend right, "h"'] \arrow[ddr, bend left, "k"] & \\
 & A' \arrow[dl, equal] \arrow[dr, equal] & \\
A' \arrow[dr, hook, "m"'] & & A' \arrow[dl, hook', "m"] \\
 & A &
\end{tikzcd}
$$

We need to prove that the spans `compSpan mf (restSpan mf)` and `mf` are in the relation `∼Span∼`. Remember that the first span is constructed as follows:

$$
\begin{tikzcd}
 & & W \arrow[dl, "h"'] \arrow[dr, "k"] & & \\
 & A' \arrow[dl, hook', "m"'] \arrow[dr, hook, "m"] & & B' \arrow[dl, hook', "m"'] \arrow[dr, "f"] & \\
A & & A & & B
\end{tikzcd}
$$

Therefore, we have to find an isomorphism between `W` and `A'` that makes the two triangles below commute.

$$
\begin{tikzcd}
& & W \arrow[dl, hook, "h"'] \arrow[d, hook, "\wr"] \arrow[dr, "k"] & & \\
& A' \arrow[dl, hook, "m"'] & A' \arrow[dll, hook, "m"] \arrow[drr, "f"'] & B' \arrow[dr, "f"] & \\
A & & & & B
\end{tikzcd}
$$

The map `h` does the job. The left diagram commutes by reflexivity. The right diagram commutes because `h` ≅ `k`, and this follows from `scom` and from `m` being a monic map. Note moreover that `h` is the unique map between the pullbacks (i) and (ii), and therefore it is an isomorphism.

```
R1Span : ∀{A B}{mf : Span A B} →
          compSpan mf (restSpan mf) ∼Span∼ mf
R1Span {mf = mf} =
  let p = proj₁ (pul∈sys (mhom mf) (m∈sys mf))
      sq' = sq p
  in record{
  s = h sq';
  sIso = pullbackIso (monicPullback (mono∈sys (m∈sys mf))) p;
  leftTr∼ = refl;
  rightTr∼ =
    cong (comp (fhom mf)) (mono∈sys (m∈sys mf) (scom sq'))}
```

The restriction `qrestSpan` is defined using `lift`, therefore, when applied to an argument `abs mf`, it is propositionally equal to `abs (restSpan mf)`. Having proved R1 up to `_∼Span∼_`, we can port this proof to `_≅_`.

```
qR1Span : ∀{A B}{x : QSpan A B} → qcompSpan x (qrestSpan x) ≅ x
qR1Span {x = x} =
  lift (λ x → qcompSpan x (qrestSpan x) ≅ x)
       (λ mf →
          proof
          qcompSpan (abs mf) (qrestSpan (abs mf))
          ≅⟨ cong (qcompSpan (abs mf)) liftbetaRest ⟩
          qcompSpan (abs mf) (abs (restSpan mf))
          ≅⟨ liftbetaComp ⟩
          abs (compSpan mf (restSpan mf))
          ≅⟨ sound R1Span ⟩
          abs mf
          ■)
        (fixtypes ∘ sound)
        x
```

The proofs of the laws R2, R3 and R4 are performed in a similar way. This completes the proof of soundness (constructing a restriction category from a partial map category).

```
RestPar : RestCat
RestPar = record{
  cat = Par;
  rest = qrestSpan;
  R1 = qR1Span;
  R2 = ?;
  R3 = ?;
  R4 = ?}
```

3.8 Idempotents

We fix a category `X`. Idempotent maps in `X` are represented as records with three fields: an object `E`, an endomap `e` on `E` and a proof `idemLaw` of `comp e e ≅ e`. Our main use of idempotents will be as objects in a category so we choose to define them as below as opposed to as a predicate on maps (see `Mono`).

```
record Idem : Set where
  field E       : Obj
        e       : Hom E E
        idemLaw : comp e e ≅ e
```

The identity map on any object is an idempotent.

```
idIdem : {A : Obj} → Idem
idIdem {A} = record{
  E = A;
  e = iden;
  idemLaw = idl}
```

A class of idempotents `IdemClass` is given primarily in terms of a membership relation (see stable systems of monics `StableSys`). The second condition states that all identities are members.

```
record IdemClass : Set where
  field ∈class   : Idem → Set
        id∈class : ∀{A} → ∈class (idIdem {A})
```

A morphism between idempotents `i` and `i'` is a map between the underlying objects paired with a proof of an equation.

```
record IdemMor (i i' : Idem) : Set where
  field imap    : Hom (E i) (E i')
        imapLaw : comp (e i') (comp imap (e i)) ≅ imap
```

Two such morphisms are equal if their underlying maps are equal. This is a consequence of uniqueness of identity proofs.

```
idemMorEq : {i i' : Idem}{f g : IdemMor i i'} →
            imap f ≅ imap g → f ≅ g
idemMorEq p = ?
```

Every morphism `f : Hom A B` in the category `X` lifts to a morphism between idempotents `idIdem{A}` and `idIdem{B}`, since `comp iden (comp f iden) ≅ f`.

```
idemMorLift : {A B : Obj}(f : Hom A B) →
              IdemMor (idIdem {A}) (idIdem {B})
idemMorLift f = record{
  imap = f;
  imapLaw =
    proof
    comp iden (comp f iden)
    ≅⟨ idl ⟩
    comp f iden
    ≅⟨ idr ⟩
    f
    ■}
```

In the proof of Lemma 2, we need the following property of a map `f` between idempotents `i` and `i'`: precomposing `imap f` with `e i` is equal to `imap f`. This is a direct consequence of the equality `imapLaw f`.

```
idemMorPrecomp : {i i' : Idem}{f : IdemMor i i'} →
                 comp (imap f) (e i) ≅ imap f
idemMorPrecomp {i}{i'}{f} =
  proof
  comp (imap f) (e i)
  ≅⟨ cong (λ y → comp y (e i)) (sym (imapLaw f)) ⟩
  comp (comp (e i') (comp (imap f) (e i))) (e i)
  ≅⟨ cong (λ y → comp y (e i)) (sym ass) ⟩
  comp (comp (comp (e i') (imap f)) (e i)) (e i)
  ≅⟨ ass ⟩
  comp (comp (e i') (imap f)) (comp (e i) (e i))
  ≅⟨ cong (comp (comp (e i') (imap f))) (idemLaw i) ⟩
  comp (comp (e i') (imap f)) (e i)
  ≅⟨ ass ⟩
  comp (e i') (comp (imap f) (e i))
  ≅⟨ imapLaw f ⟩
  imap f
  ■
```

Analogously, one can prove that postcomposing `imap f` with `e i'` is equal to `imap f`.

```
idemMorPostcomp : {i i' : Idem}{f : IdemMor i i'} →
                  comp (e i') (imap f) ≅ imap f
idemMorPostcomp {i}{i'} f =
  proof
  comp (e i') (imap f)
  ≅⟨ cong (comp (e i')) (sym (imapLaw f)) ⟩
  comp (e i') (comp (e i') (comp (imap f) (e i)))
  ≅⟨ sym ass ⟩
  comp (comp (e i') (e i')) (comp (imap f) (e i))
  ≅⟨ cong (λ y → comp y (comp (imap f) (e i))) (idemLaw i') ⟩
```

```
  comp (e i') (comp (imap f) (e i))
  ≅⟨ imapLaw f ⟩
  imap f
  ■
```

Idempotents and morphisms between them form a category. Identities are given by the idempotents themselves.

```
idIdemMor : {i : Idem} → IdemMor i i
idIdemMor {i} = record{
  imap = e i;
  imapLaw =
    proof
    comp (e i) (comp (e i) (e i))
    ≅⟨ cong (comp (e i)) (idemLaw i) ⟩
    comp (e i) (e i)
    ≅⟨ idemLaw i ⟩
    e i
    ■}
```

Composition is inherited from the underlying category.

```
compIdemMor : {i₁ i₂ i₃ : Idem}
              (g : IdemMor i₂ i₃)(f : IdemMor i₁ i₂) →
              IdemMor i₁ i₃
compIdemMor {i₁}{i₂}{i₃} g f = record{
  imap = comp (imap g) (imap f);
  imapLaw =
    proof
    comp (e i₃) (comp (comp (imap g) (imap f)) (e i₁))
    ≅⟨ cong (comp (e i₃)) ass ⟩
    comp (e i₃) (comp (imap g) (comp (imap f) (e i₁)))
    ≅⟨ cong (λ y → comp (e i₃) (comp (imap g) y)) idemMorPrecomp ⟩
    comp (e i₃) (comp (imap g) (imap f))
    ≅⟨ sym ass ⟩
    comp (comp (e i₃) (imap g)) (imap f)
    ≅⟨ cong (λ y → comp y (imap f)) idemMorPostcomp ⟩
    comp (imap g) (imap f)
    ■}
```

The associativity law follows directly from the associativity law of the underlying category. The identity laws do not follow directly, since identities in this new category are idempotents, but they are immediate consequences of `idemMorPrecomp` and `idemMorPostcomp`.

```
SplitCat : IdemClass → Cat
SplitCat E = record{
  Obj = Σ Idem (∈class E);
  Hom = λ ip jq → IdemMor (proj₁ ip) (proj₁ jq);
  iden = idIdemMor;
```

```
  comp = compIdemMor;
  idl = idemMorEq idemMorPostcomp;
  idr = idemMorEq idemMorPrecomp;
  ass = idemMorEq ass}
```

Given a class of idempotents `E`, our category `X` is a full subcategory of `SplitCat E`. We define the inclusion functor `InclSplitCat`. It sends an object `A` to its corresponding identity `idIdem {A}`, which belongs to `E` by definition of class of idempotents, and it sends a morphism `f` to its lifting `idemMorLift f`. The functor laws hold trivially.

```
InclSplitCat : (E : IdemClass) → Fun X (SplitCat E)
InclSplitCat E = record{
  OMap  = λ A → idIdem {A} , id∈class E;
  HMap  = idemMorLift;
  fid   = idemMorEq refl;
  fcomp = idemMorEq refl}
```

Since `InclSplitCat` is basically identity on morphisms, it is easy to show that it is a full and faithful functor.

```
FullInclSplitCat : {E : IdemClass} → Full (InclSplitCat E)
FullInclSplitCat {f = f} = imap f , idemMorEq refl

FaithfulInclSplitCat : {E : IdemClass} → Faithful (InclSplitCat E)
FaithfulInclSplitCat refl = refl
```

Moreover, the category `SplitCat E` is a restriction category, if the original category `X` is a restriction category. So let `X` be a restriction category and `Xcat` its underlying category. We describe formally the restriction operation on `SplitCat E`. Given a morphism `f : IdemMor i i'` in `SplitCat E`, the restriction of `f` has `comp (rest (imap f)) (e i)` as underlying morphism in `Xcat` (this corresponds to the 'hat' operation described in Lemma 2).

```
restIdemMor : {i i' : Idem} → IdemMor i i' → IdemMor i i
restIdemMor {i} f = record{
  imap = comp (rest (imap f)) (e i);
  imapLaw =
    proof
    comp (e i) (comp (comp (rest (imap f)) (e i)) (e i))
    ≅⟨ cong (comp (e i)) ass ⟩
    comp (e i) (comp (rest (imap f)) (comp (e i) (e i)))
    ≅⟨ cong (comp (e i) ∘ comp (rest (imap f))) (idemLaw i) ⟩
    comp (e i) (comp (rest (imap f)) (e i))
    ≅⟨ cong (comp (e i)) R4 ⟩
    comp (e i) (comp (e i) (rest (comp (imap f) (e i))))
    ≅⟨ sym ass ⟩
    comp (comp (e i) (e i)) (rest (comp (imap f) (e i)))
    ≅⟨ cong (λ y → comp y (rest (comp (imap f) (e i))))
            (idemLaw i) ⟩
```

```
    comp (e i) (rest (comp (imap f) (e i)))
    ≅⟨ sym R4 ⟩
    comp (rest (imap f)) (e i)
    ■}
```

The restriction category axioms are easily provable. For example, R1 is a direct consequence of `X` being a restriction category and the property `idemMorPrecomp`. Remember that two parallel morphisms in `SplitCat E` are equal, if their underlying maps in `Xcat` are equal (a property we named `idemMorEq`).

```
R1Split : {E : IdemClass}{ip jq : Σ Idem (∈class E)}
          {f : IdemMor (proj₁ ip) (proj₁ jq)} →
          compIdemMor f (restIdemMor f) ≅ f
R1Split {ip = i , p}{f = f} =
  idemMorEq
    (proof
     comp (imap f) (comp (rest (imap f)) (e i))
     ≅⟨ sym ass ⟩
     comp (comp (imap f) (rest (imap f))) (e i)
     ≅⟨ cong (λ y → comp y (e i)) R1 ⟩
     comp (imap f) (e i)
     ≅⟨ idemMorPrecomp ⟩
     imap f
     ■)
```

Proofs for R2, R3 and R4 are performed in a similar way.

```
RestSplitCat : (E : IdemClass) → RestCat
RestSplitCat E = record{
  cat = SplitCat E;
  rest = restIdemMor;
  R1 = R1Split;
  R2 = ?;
  R3 = ?;
  R4 = ?}
```

Lemma 2 can now be proved. Any restriction category `X` embeds fully in the restriction category `RestSplitCat E` for any class of idempotents `E`.

```
InclRestSplitCat : (E : IdemClass) → RestFun X (RestSplitCat E)
InclRestSplitCat E = record{
  fun = InclSplitCat E;
  frest = idemMorEq idr}
```

3.9 Restriction Idempotents

We fix a restriction category `X` with underlying category `Xcat`. We define a predicate `isRestIdem` stating that an idempotent is a restriction idempotent. An idempotent is a restriction idempotent, if it is equal to its restriction.

```
isRestIdem : Idem → Set
isRestIdem i = e i ≅ rest (e i)
```

Restriction idempotents define a class of idempotents `restIdemClass`. Identity maps belong to the class, since they are monic (`idMono`) and monic maps are total (`lem1`).

```
restIdemClass : IdemClass
restIdemClass = record{
  ∈class = isRestIdem;
  id∈class = sym (lem1 idMono)}
```

A splitting of an idempotent `i` on an object `E` is a record consisting of an object `B`, a section from `B` to `E`, a retraction from `E` to `B` and proofs of two equations.

```
record Split (i : Idem) : Set where
  field B         : Obj
        sec       : Hom B (E i)
        retr      : Hom (E i) B
        splitLaw1 : comp sec retr ≅ e i
        splitLaw2 : comp retr sec ≅ iden {B}
```

A restriction category where all restriction idempotents are split is called a split restriction category.

```
record SplitRestCat : Set where
  field rcat          : RestCat
        restIdemSplit : (i : Idem (cat rcat)) →
                        isRestIdem rcat i → Split (cat rcat) i
```

Lemma 3 states that the restriction category `RestSplitCat restIdemClass` (built from a restriction category `X`) is a split restriction category. For readability and simplicity reasons, we do not show the proof that every restriction idempotent is split.

```
SplitRestSplitCat : SplitRestCat
SplitRestSplitCat = record{
  rcat = RestSplitCat restIdemClass;
  restIdemSplit = ?}
```

3.10 Completeness

In order to state the completeness theorem (Theorem 2), we have to construct the stable system of monics `SectionsOfRestIdem` given by the sections of the restriction idempotents of a particular split restriction category. We fix a split restriction category `X` with underlying restriction category `Xrcat` and underlying category `Xcat`. First we define a record `SectionOfRestIdem` parametrized by a total map. The proposition `SectionOfRestIdem s` holds if and only if `hom s` is a section of a restriction idempotent.

```
record SectionOfRestIdem {B E} (s : Tot B E) : Set where
  field e         : Hom E E
        restIdem  : e ≅ rest e
        r         : Hom E B
        splitLaw1 : comp (hom s) r ≅ e
        splitLaw2 : comp r (hom s) ≅ iden {B}
```

The predicate `SectionOfRestIdem` defines a stable system of monics in the subcategory of total maps in `rcat`. We show that every isomorphism is a section of a restriction idempotent. We do not show the proof that every map in the system is monic and the proofs that the system is closed under composition and pullback. Note that identity maps are restriction idempotents and every isomorphism is the section of an identity map.

```
iso∈sysSectionOfRestIdem : ∀{B E}{s : Tot B E} → Iso s →
                            SectionOfRestIdem s
iso∈sysSectionOfRestIdem i = record{
  e = iden;
  restIdem = sym (lem1 (idMono cat));
  r = hom (inv i);
  splitLaw1 = cong hom (rinv i);
  splitLaw2 = cong hom (linv i)}

SectionOfRestIdemSys : StableSys Total
SectionOfRestIdemSys = record{
  ∈sys = SectionOfRestIdem;
  mono∈sys = ?;
  iso∈sys = iso∈sysSectionOfRestIdem;
  comp∈sys = ?;
  pul∈sys = ?}
```

We now move to the formalization of the completeness theorem (Theorem 2) for a particular split restriction category `X` with underlying restriction category `Xrcat` and underlying category `Xcat`. Let `Par` be the partial map category over the category of total maps `Total` and stable system of monics `SectionsOfRestIdemSys`, and `RestPar` the restriction category on top of `Par` given by soundness. We show the construction of the functors `Funct : Fun Xcat Par` and `Funct2 : Fun Par Xcat`. These functors can be lifted to restriction functors `RFunct` and `RFunct2` and they are each other inverses in the category of restriction categories and restriction functors, therefore showing that `Xrcat` and `RestPar` are isomorphic in this category.

The functor `Funct` is identity on objects. The mapping of maps of the functor `Funct` takes a map `f : Hom A C` in `Xcat` and returns a map in `Par`, i.e., an element of `QSpan A C`. We first define a function `HMap1` that constructs a span between `A` and `C`. The mapping of maps of `Funct` will be `abs ∘ HMap1`. The map `rest f` is a an idempotent (`lem2`), moreover a restriction idempotent (`lem3`), therefore it splits.

```
restIdemIdemGen : ∀{A C}(f : Hom A C) → Idem
restIdemIdemGen {A} f = record{
  E = A;
  e = rest f;
  idemLaw = lem2}

restIdemSplitGen : ∀{A C}(f : Hom A C) → Split (restIdemIdemGen f)
restIdemSplitGen f = restIdemSplit (restIdemIdemGen f) (sym lem3)
```

For the left leg of the span, we take the section `sec (restIdemSplitGen f)`, which is total. For the right leg, we take the map `comp f (sec (restIdemSplitGen f))`, which is also total.

```
leftLeg : ∀{A C}(f : Hom A C) → Tot (B (restIdemSplitGen f)) A
leftLeg f = record{
  hom = sec (restIdemSplitGen f);
  totProp = ?}

rightLeg : ∀{A C}(f : Hom A C) → Tot (B (restIdemSplitGen f)) C
rightLeg f = record{
  hom = comp f (sec (restIdemSplitGen f));
  totProp = ?}
```

This concludes the definition of the functor `Funct`. The total map `leftLeg f` is the section of a restriction idempotent, i.e., the type `SectionOfRestIdem (leftLeg f)` is inhabited.

```
HMap1 : ∀{A C}(f : Hom A C) → Span A C
HMap1 f = record{
  A' = B (restIdemSplitGen f);
  mhom = leftLeg f;
  fhom = rightLeg f;
  m∈sys = ?}
```

`HMap1` is required to preserve identities and composition. Here we show that it preserves composition up to `_∼Span∼_`. Let `f : Hom A C` and `g : Hom C D`. We prove `HMap1 (comp g f) ∼Span∼ compSpan (HMap1 g) (HMap1 f)`. Since the restriction idempotents split, in particular we have the following three diagrams.

$$
\begin{tikzcd} A \arrow[dr, "r_1"] \arrow[dd, "\overline{f}"'] & \\ & B_1 \arrow[dl, hook, "s_1"] \\ A & \end{tikzcd}
\qquad
\begin{tikzcd} C \arrow[dr, "r_2"] \arrow[dd, "\overline{g}"'] & \\ & B_2 \arrow[dl, hook, "s_2"] \\ C & \end{tikzcd}
\qquad
\begin{tikzcd} A \arrow[dr, "r_3"] \arrow[dd, "\overline{g \circ f}"'] & \\ & B_3 \arrow[dl, hook, "s_3"] \\ A & \end{tikzcd}
$$

The span `HMap1 (comp g f)` is

$$
\begin{tikzcd} & B_3 \arrow[dl, hook, "s_3"'] \arrow[dr, "g \circ f \circ s_3"] & \\ A & & D \end{tikzcd}
$$

while the span `compSpan (HMap1 g) (HMap1 f)` is

$$
\begin{tikzcd} & & W \arrow[dl, hook, "h"'] \arrow[dr, "k"] \arrow[dd, phantom, very near start, "\lrcorner" rotate=-45] & & \\ & B_1 \arrow[dl, hook, "s_1"'] \arrow[dr, "f \circ s_1"] & & B_2 \arrow[dl, hook, "s_2"'] \arrow[dr, "g \circ s_2"] & \\ A & & C & & D \end{tikzcd}
$$

Notice that the map `h` is in the stable system of monics, i.e. it is the section of a restriction idempotent. This is true because `h` is the pullback of $\mathtt{s}_2$, which is in the stable system of monics. In particular, there exists a restriction idempotent `w : Hom W W` and a map `r : Hom` B_1 `W` such that the following triangle commutes.

$$\begin{tikzcd} B_1 \arrow[d, "w"'] \arrow[dr, "r"] & \\ B_1 & W \arrow[l, hook, "h"] \end{tikzcd}$$

Our goal is to find an isomorphism `u : Hom` B_3 `W` that makes the two generated triangles commute. It is not difficult to show that the composite map

$$u = B_3 \xrightarrow{s_3} A \xrightarrow{r_1} B_1 \xrightarrow{r} W$$

does the job. As usual, we refer to the Agda formalization for more details.

We obtain a functor `Funct` between the categories `Xcat` and `Par`.

```
Funct : Fun Xcat Par
Funct = record{
  OMap = id;
  HMap = abs ∘ HMap1;
  fid = ?;
  fcomp = ?}
```

The functor `Funct` also preserves the restriction operation. Therefore it is a restriction functor.

```
RFunct : RestFun Xrcat RestPar
RFunct = record{
  fun = Funct;
  frest = ?}
```

The functor `Funct2` is also identity on objects. The mapping of maps of the functor `Funct2` takes an element of `QSpan A C` into a map between `A` and `C`. We first define a function `HMap2` from `Span A C` into `Hom A C`. We fix a span `mf`. Let `A'` the object, `m` and `f` be the left and right legs of `mf` (which are total maps), and `m∈` be the proof that `m` is in the stable system of monics, i.e., `m∈` states that the total map `m` is the section of a restriction idempotent. The morphisms `hom f : Hom A' C` and `r m∈ : Hom A A'` are composable, and their composition defines `HMap2`.

```
HMap2 : ∀{A C} → Span A C → Hom A C
HMap2 mf = comp (hom (fhom mf)) (r (m∈sys mf))
```

`HMap2` is compatible with the equivalence relation `∼Span∼` on `Span A C`. So it can be lifted to a function `qHMap2` on the quotient `QSpan A C`. This concludes the description of the functor `Funct2`.

```
qHMap2 : ∀{A C} → QSpan A C → Hom A C
qHMap2 {A}{C} = lift {A}{C} HMap2 ?
```

The function `qHMap2` is propositionally equal to `HMap2 mf`, when applied to a term `abs nm`.

```
liftbetaqHMap2 : ∀{A C}{mf : Span A C} → qHMap2 (abs mf) ≅ HMap2 mf
liftbetaqHMap2 = liftbeta _ HMap2 ? _
```

It is not difficult to see that `qHMap2` preserves identities and composition. We obtain a functor `Funct2` between `Par` and `Xcat`.

```
Funct2 : Fun Par Xcat
Funct2 = record{
  OMap = id;
  HMap = qHMap2;
  fid = ?;
  fcomp = ?}
```

The functor `Funct2` preserves the restriction operation. Therefore it is a restriction functor.

```
RFunct2 : RestFun RestPar Xrcat
RFunct2 = record{
  fun = Funct2;
  frest = ?}
```

The functors `RFunct` and `RFunct2` are each other inverses. First, let `mf : Span A C`. We show that `HMap1 (HMap2 mf) ~Span~ mf`. Let `m : Hom` A_1 `A` be the left leg of `mf` and `f : Hom` A_1 `C` the right leg. The map `m` is the section of a restriction idempotent. It is possible to prove that it is the section of `rest` r_1, where r_1 is the retraction of the splitting. In particular, the following diagram commutes.

$$
\begin{tikzcd}
A \arrow[dr, two heads, "r_1"] \arrow[dd, "\overline{r_1}"'] & \\
& A_1 \arrow[dl, hook', "m"] \\
A &
\end{tikzcd}
$$

Let `n : Hom` A_2 `A` be the left leg of `HMap1 (HMap2 mf)`, the right leg is `comp (comp f` r_1`) n` by construction. The map `n` is the section of the restriction idempotent `rest (comp f` r_1`)`, and the latter is equal to `rest` r_1 because `f` is total. In particular, there exists a map r_2 `: Hom A` A_2 making the following diagram commute.

$$
\begin{tikzcd}
A \arrow[dr, two heads, "r_2"] \arrow[dd, "\overline{r_1}"'] & \\
& A_2 \arrow[dl, hook', "n"] \\
A &
\end{tikzcd}
$$

It is not difficult to prove that the map `comp` r_1 `n : Hom` A_2 A_1 is an isomorphism between the spans `HMap1 (HMap2 mf)` and `mf`. This construction lifts straightforwardly to the the quotient `QSpan A C`.

```
HIso1 : ∀{A C}(mf : QSpan A C) → abs (HMap1 (qHMap2 mf)) ≅ mf
HIso1 mf = ?
```

On the other hand, consider a map `f : Hom A C`. The map `HMap2 (HMap1 f)` is given by `comp (comp f s) r`, where `s` and `r` are the section and the retraction of the splitting of `rest f`.

```
HIso2 : ∀{A C}(f : Hom A C) → qHMap2 (abs (HMap1 f)) ≅ f
HIso2 f =
  let open Split (restIdemSplitGen f)
  in
    proof
    qHMap2 (abs (HMap1 f))
    ≅⟨ qHMap2Liftbeta ⟩
    comp (comp f s) r
    ≅⟨ ass ⟩
    comp f (comp s r)
    ≅⟨ cong (comp f) splitLaw1 ⟩
    comp f (rest f)
    ≅⟨ R1 ⟩
    f
    ■
```

This completes the proof of Theorem 2: every split restriction category is isomorphic to a partial map category in the category of restriction categories and restriction functors.

4. CONCLUSION AND FUTURE WORK

We formalized in Agda the first chapters of the theory of restriction categories and learned that this was overall a feasible project. In an earlier version of this paper, we used a more primitive approach to quotients that did not require us to show that functions from quotients respect equality. The version of quotients presented here and used in the full formalization does require this and we thank an anonymous referee for suggesting this approach.

Formalization of category theory requires extensive use of record types. We believe that we exploited the various features of Agda's current design (especially the idea that records are modules) quite well, although there is probably room for further improvement in our code with regards to modularity.

We plan to extend this work to cover joins and meets of maps in restriction categories, restriction products, iteration, Turing categories, and partial map classifiers.

We will also link it to programming with partial functions in DTP. Namely, we will elaborate specific examples of restriction categories, first of all the Kleisli category of Capretta's delay monad, which is the constructive alternative to the maybe monad.

In this direction, we have already formalized [5] basic facts about the delay monad with Hofmann's inductive-like quotient types. An interesting issue arises—the axiom of countable choice is needed to define the multiplication of the monad, if one works with quotient types instead of just setoids.

In a forthcoming article, we will show that the delay monad is an equational lifting monad in the sense of Bucalo et al. [3]. By the results of Cockett and Lack [8, 9], its Kleisli category is therefore a restriction category. The delay monad delivers free ω-cppos. It is initial among those monads whose Kleisli category is equipped with countable join restriction structure.

Acknowledgements. We thank Robin Cockett for teaching us about restriction categories and Silvio Capobianco for producing a comprehensive set of notes. We are grateful to the anonymous reviewers of the paper at the different stages of its making.

References

[1] A. Bove and V. Capretta. Modelling general recursion in type theory. *Math. Struct. Comput. Sci.*, 15(4):671–708, 2005.

[2] A. Bove, A. Krauss, and M. Sozeau. Partiality and recursion in interactive theorem provers: An overview. *Math. Struct. Comput. Sci.*, 26(1):38–88, 2016.

[3] A. Bucalo, C. Führmann, and A. Simpson. An equational notion of lifting monad. *Theor. Comput. Sci.*, 294(1–2):31–60, 2003.

[4] V. Capretta. General recursion via coinductive types. *Log. Meth. Comput. Sci.*, 1(2), 2005.

[5] J. Chapman, T. Uustalu, and N. Veltri. Quotienting the delay monad by weak bisimilarity. In M. Leucker, C. Rueda, and F. D. Valencia, editors, *Proc. of 12th Int. Coll. on Theoretical Aspects of Computing, ICTAC 2015*, volume 9399 of *Lect. Notes in Comput. Sci.*, pages 110–125. Springer, 2015.

[6] J. R. B. Cockett and P. J. W. Hofstra. Introduction to Turing categories. *Ann. Pure Appl. Logic*, 156(2–3):183–209, 2008.

[7] J. R. B. Cockett and S. Lack. Restriction categories I: categories of partial maps. *Theor. Comput. Sci.*, 270(1–2):223–259, 2002.

[8] J. R. B. Cockett and S. Lack. Restriction categories II: partial map classification. *Theor. Comput. Sci.*, 294(1–2):61–102, 2003.

[9] J. R. B. Cockett and S. Lack. Restriction categories III: colimits, partial limits and extensivity. *Math. Struct. Comput. Sci.*, 17(4):775–817, 2007.

[10] R. Cockett, J. Díaz-Boïls, J. Gallagher, and P. Hrubeš. Timed sets, functional complexity, and computability. *Electron. Notes Theor. Comput. Sci.*, 286:117–137, 2012.

[11] R. A. Di Paola and A. Heller. Dominical categories: Recursion theory without elements. *J. Symb. Log.*, 52(3):594–635, 1987.

[12] M. Hofmann. *Extensional Concepts in Intensional Type Theory.* CPHS/BCS Distinguished Dissertations. Springer, 1997.

[13] K. Menger. An axiomatic theory of functions and fluents. In L. Henkin, P. Suppes, and A. Tarski, editors, *The Axiomatic Method*, volume 27 of *Studies in Logic and the Foundations of Mathematics*, pages 454–473. North-Holland, 1959.

[14] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Revised Lecture Notes from 6th Int. School on Advanced Functional Programming, AFP 2008*, volume 5832 of *Lect. Notes in Comput. Sci.*, pages 230–266. Springer, 2009.

[15] E. Robinson and G. Rosolini. Categories of partial maps. *Inf. Comput.*, 79(2):95–130, 1988.