Formal verification in Coq of program properties involving the global state effect

Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, Damien Pous

Formal verification in Coq of program properties involving the global state effect

Jean-Guillaume Dumas* Dominique Duval* Burak Ekici*

Damien Pous†

October 2, 2013

Abstract

The syntax of an imperative language does not mention explicitly the state, while its denotational semantics has to mention it. In this paper we present a framework for the verification in Coq of properties of programs manipulating the global state effect. These properties are expressed in a proof system which is close to the syntax, as in effect systems, in the sense that the state does not appear explicitly in the type of expressions which manipulate it. Rather, the state appears via decorations added to terms and to equations. In this system, proofs of programs thus present two aspects: properties can be verified *up to effects* or the effects can be taken into account. The design of our Coq library consequently reflects these two aspects: our framework is centered around the construction of two inductive and dependent types, one for terms up to effects and one for the manipulation of decorations.

1 Introduction

The evolution of the state of the memory in an imperative program is a computational effect: the state is never mentioned as an argument or a result of a command, whereas in general it is used and modified during the execution of commands. Thus, the syntax of an imperative language does not mention explicitly the state, while its denotational semantics has to mention it. This means that the state is encapsulated: its interface, which is made of the functions for looking up and updating the values of the locations, is separated from its implementation; the state cannot be accessed in any other way than through his interface.

It turns out that equational proofs in an imperative language may also encapsulate the state: proofs can be performed without any knowledge of the implementation of the state. This is made possible by adding decorations to terms, as in effect-systems [6, 11], and by also decorating the equations [3]. The latter approach uses category theoretical constructions to model the denotational semantics of the state effect and prove some properties of programs involving this effect. *Strong monads*, introduced by Moggi [7], were the first categorical approach to computational effects, while Power et al [9] then proposed the *premonoidal categories*. Next Hughes [5] extended Haskell with arrows

*LJK, Université de Grenoble, France. \protect\protect\T1\textbraceleftJean-Guillaume.Dumas,Dominique.Duval,Burak.Ekici\pro

†LIP, ENS Lyon, France. Damien.Pous@ens-lyon.fr

that share some properties with the approach of *cartesian effect categories* of Dumas et al [4].

The goal of this paper is to propose a Coq environment where proofs, written in the latter decorated framework for the state effect, could be automatically verified.

Proving properties of programs involving the state effect is important when the order of evaluation of the arguments is not specified or more generally when parallelization comes into play [6]. Indeed, pure computations, i.e. those not having any side-effects (or in other words not modifying the state), are independent could thus be run in parallel. Differently, computations depending on or modifying the state should be handled with more care.

Now, proofs involving side-effects can become quite complex in order to be fully rigorous. We will for instance look at the following property in details: *recovering the value of a variable and setting up the value of another variable can be performed in any order*. Such properties have been formalized for instance by Plotkin et al [8] but the full mathematical proof of such properties can be quite large. The decorated approach of [3] helps since it enables a verification of such proofs in two steps: a first step checks the syntax *up to effects* by dropping the decorations; a second step then takes the effects into account.

To some extent, our work looks quite similar to the work by [2] in the sense that we also define our own programming language and verify its properties by using axiomatic semantics. We construct our system on category theoretical notions (e.g. monads) as done in [1]. In brief, we first declare our system components including their properties and then prove some related propositions. In that manner, the overall idea is also quite close to the one given in [10], even though technical details completely differ.

In this paper, we show that the latter decorated proof system can be developed in Coq thus enabling an automatic verification of decorated proofs for side-effect systems. We recall in Section 2 the logical environment for decorated equational proofs involving the state effect. Then in Section 3 we present the translation of the categorical rules into Coq as well as their resulting derivations and the necessary additions. The resulting Coq code has been integrated into a library, available there: `http://coqeffects.forge.imag.fr`. Finally, in Section 4 we give the full details of the proof of the property above and its verification in Coq, as an example of the capabilities of our library. Appendix A is then added for the sake of completeness and readability in order to give the logical counterparts of the rules verified in our Coq library.

2 The Logical Environment for Equational Proofs

2.1 Motivation

Basically, in a purely functional programming language, an operation or a term f with an argument of type X and a result of type Y, which may be written $f : X \to Y$ (in the *syntax*), is interpreted (in the *denotational semantics*) as a function $[\![f]\!]$ between the sets $[\![X]\!]$ and $[\![Y]\!]$, interpretations of X and Y. It follows that, when an operation has several arguments, these arguments can be evaluated in parallel, or in any order. It is possible to interpret a purely functional programming language via a categorical semantics based on *cartesian closed categories*; the word "cartesian" here refers to the categorical *products*, which are interpreted as *cartesian products* of sets, and which are used for dealing with pairs (or tuples) of arguments. The *logical semantics* of the

language defines a set of rules that may be used for proving properties of programs.

But non-functional programming languages such as `C` or `Java` do include computational effects. For instance a `C` function may modify the state structure and a `Java` function may throw an exception during the computation. Such operations are examples of computational effects. In this paper we focus on the states effect. We consider the *lookup* and *update* operations for modeling the behavior of imperative programs: namely an *update* operation assigns a value to a location (or variable) and a *lookup* operation recovers the value of a location. There are many ways to handle computational effects in programming languages. Here we focus on the categorical treatment of [4], adapted to the state effect [3]: this provides a logical semantics relying on *decorations*, or annotations, of terms and equations.

2.2 Decorated functions and equations for the states effect

The functions in our language

are classified according to the way they interact with the state. The classification takes the form of annotations, or decorations, written as superscripts. A function can be a *modifier*, an *accessor* or a *pure* function.

- As the name suggests, a *modifier* may modify or use the state: it is a *read-write* function. We will use the keyword *rw* as an annotation for modifiers.
- An *accessor* may use the state structure but never modifies it: it is a *read-only* function. We will use the keyword *ro* for accessors.
- A *pure function* never interacts with the state. We will use the keyword *pure* for pure functions.

The denotational semantics of this language is given in terms of the set of states S and the *cartesian product* operator '$\times$'. For all types X and Y, interpreted as sets $[\![X]\!]$ and $[\![Y]\!]$, a modifier function $f : X \to Y$ is interpreted as a function $[\![f]\!] : [\![X]\!] \times S \to [\![Y]\!] \times S$ (it can access the state and modify it); an accessor g as $[\![g]\!] : [\![X]\!] \times S \to [\![Y]\!]$ (it can access the state but not modify it); and a pure function h as $[\![h]\!] : [\![X]\!] \to [\![Y]\!]$ (it can neither access nor modify the state). There is a hierarchy among those functions. Indeed any pure function can be seen as both an accessor or a modifier even though it will actually do not make use of its argument S. Similarly an accessor can be seen as a modifier.

The state is made of memory *locations*, or *variables*; each location has a value which can be updated. For each location i, let V_i be the type of the values that can be stored in the location i, and let $Val_i = [\![V_i]\!]$ be the interpretation of V_i. In addition, the unit type is denoted by $\mathbb{1}$; its interpretation is a singleton, it will also be denoted by $\mathbb{1}$.

The assignment of a value of type V_i to a variable i takes an argument of type V_i. It does not return any result but it modifies the state: given a value $a \in Val_i$, the assignment of a to i sets the value of location i to a and keeps the value of the other locations unchanged. Thus, this operation is a modifier from V_i to $\mathbb{1}$. It is denoted by $update_i^{rw} : V_i \to \mathbb{1}$ and it is interpreted as $[\![update_i]\!] : Val_i \times S \to S$.

The recovery of the value stored in a location i takes no argument an returns a value of type V_i. It does not modify the state but it observes the value stored at location i. Thus, this operation is an accessor from $\mathbb{1}$ to V_i. It is denoted by $lookup_i^{ro} : \mathbb{1} \to V_i$ and it is interpreted (since $\mathbb{1} \times S$ is in bijection with S) as $[\![lookup_i]\!] : S \to Val_i$.

For each type X, the *identity* operation $id_X : X \to X$, which is interpreted by mapping each element of $[\![X]\!]$ to itself, is pure.

Similarly, the *final* operation $\langle\rangle_X : X \to \mathbb{1}$, which is interpreted by mapping each element of $[\![X]\!]$ to the unique element of the singleton $\mathbb{1}$, is pure. In order to lighten the notations we will often use id_i and $\langle\rangle_i$ instead of respectively id_{Val_i} and $\langle\rangle_{Val_i}$.

In addition, decorations are also added to equations.

- Two functions $f, g : X \to Y$ are *strongly equal* if they return the same result and have the same effect on the state structure. This is denoted $f == g$.
- Two functions $f, g : X \to Y$ are *weakly equal* if they return the same result but may have different effects on the state. This is denoted $f \sim g$.

The state can be observed thanks to the lookup functions. For each location i, the interpretation of the $update_i$ operation is characterized by the following equalities, for each state $s \in S$ and each $x \in Val_i$:

$$\begin{cases} [\![lookup_i]\!]([\![update_i]\!](s,x)) = x \\ [\![lookup_j]\!]([\![update_i]\!](s,x)) = [\![lookup_j]\!](s) \text{ for every } j \in Loc,\ j \neq i \end{cases}$$

According to the previous definitions, these equalities are the interpretations of the following weak equations:

$$\begin{cases} lookup_i^{ro} \circ update_i^{rw} \sim id_i^{pure} : V_i \to V_i \\ lookup_j^{ro} \circ update_i^{rw} \sim lookup_j^{ro} \circ \langle\rangle_i^{pure} \text{ for every } j \in Loc,\ j \neq i : V_i \to V_j \end{cases}$$

2.3 Sequential products

In functional programming, the product of functions allows to model operations with several arguments. But when side-effects occur (typically, updates of the state), the result of evaluating the arguments may depend on the order in which they are evaluated. Therefore, we use *sequential products* of functions, as introduced in [4], which impose some order of evaluation of the arguments: a sequential product is obtained as the sequential composition of two *semi-pure products*. A semi-pure product, as far as we are concerned in this paper, is a kind of product of an identity function (which is pure) with another function (which may be any modifier).

For each types X and Y, we introduce a *product* type $X \times Y$ with *projections* $\pi^{pure}_{1,X_1,X_2} : X_1 \times X_2 \to X_1$ and $\pi^{pure}_{2,X_1,X_2} : X_1 \times X_2 \to X_2$, which will be denoted simply by π_1^{pure} and π_2^{pure}. This is interpreted as the cartesian product with its projections. Pairs and products of pure functions are built as usual. In the special case of a product with the unit type, it can easily be proved, as usual, that $\pi_1^{pure} : X \times \mathbb{1} \to X$ is invertible with inverse the pair $(\pi_1^{-1})^{pure} = \langle id_X^{pure}, \langle\rangle_X^{pure}\rangle : X \to X \times \mathbb{1}$, and that $\pi_2^{pure} = \langle\rangle_X^{pure} : X \times \mathbb{1} \to \mathbb{1}$. The *permutation* operation $perm_{X\times Y} : X \times Y \to Y \times X$ is also pure: it is interpreted as the function which exchanges its two arguments.

Given a modifier $f^{rw} : X \to Y$ and its interpretation $[\![f]\!] : [\![X]\!] \times S \to [\![Y]\!] \times S$, the *left semi-pure pair* $\langle id_X, f\rangle_l^{rw} : X \to X \times Y$ is the modifier interpreted by $[\![\langle id_X, f\rangle_l]\!] : [\![X]\!] \times S \to [\![X]\!] \times [\![Y]\!] \times S$ such that $[\![\langle id_X, f\rangle_l]\!](x,s) = (x,y,s')$ where $(y,s') = [\![f]\!](x,s)$. This is a generalization of the usual *graph* of a function. The left semi-pure pair $\langle id_X, f\rangle_l^{rw}$ is characterized, up to strong equations, by a weak and a strong equation:

$$\pi_1^{pure} \circ \langle id_X, f\rangle_l^{rw} \sim id_X^{pure} \text{ and } \pi_2^{pure} \circ \langle id_X, f\rangle_l^{rw} == f^{rw}$$

The *right semi-pure pair* $\langle f, id_X\rangle_r^{rw} : X \to Y \times X$ is defined in the symmetric way. This is illustrated in Figure 1.

Note. In all diagrams, the decorations are expressed by shapes and colors of arrows: waving arrows for pure functions, black and red (or grey in B/W viewing) straight arrows for accessors and modifiers, respectively.

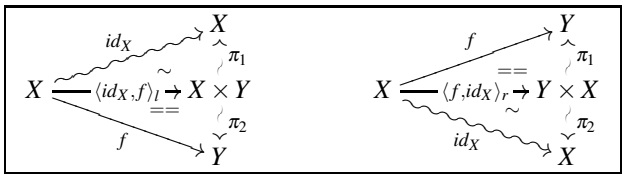

Figure 1: Left and right semi-pure pairs

The *left semi-pure product* is defined in the usual way from the left semi-pure pair: given $f^{rw}\colon X \to Y$ and a type Z, the left semi-pure product of id_Z and f is $(id_Z \ltimes f)^{rw} = \langle \pi_{1,Z,X}, f \circ \pi_{2,Z,X}\rangle_l^{rw}\colon Z \times X \to Z \times Y$. It is characterized, up to strong equations, by a weak and a strong equation:

$$\pi^{pure}_{1,Z,Y} \circ (id_Z \ltimes f)^{rw} \sim \pi^{pure}_{1,Z,X} \text{ and } \pi^{pure}_{2,Z,Y} \circ (id_Z \ltimes f)^{rw} == f^{rw} \circ \pi^{pure}_{2,Z,X}$$

This means that the "context" in Z is kept unchanged while the modifier f is executed. The *right semi-pure product* $(f \rtimes id_Z)^{rw}\colon X \times Z \to Y \times Z$ is defined in the symmetric way. This is illustrated in Figure 2.

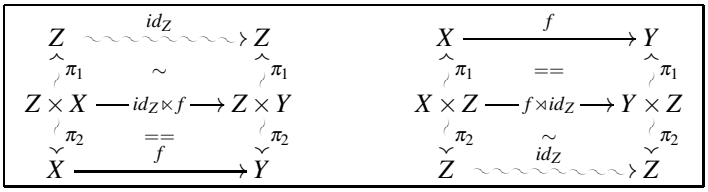

Figure 2: Left and right semi-pure products

Now, it is easy to define the *left sequential product* of two modifiers $f_1^{rw} : X_1 \to Y_1$ and $f_2^{rw} : X_2 \to Y_2$ by composing a right semi-pure product with a left semi-pure one, as follows:

$$(f_1 \ltimes f_2)^{rw} = (id_{Y_1} \ltimes f_2)^{rw} \circ (f_1 \rtimes id_{X_2})^{rw}\colon X_1 \times X_2 \to Y_1 \times Y_2$$

In a symmetric way, the *right sequential product* of $f_1^{rw} : X_1 \to Y_1$ and $f_2^{rw} : X_2 \to Y_2$ is defined as:

$$(f_1 \rtimes f_2)^{rw} = (f_1 \rtimes id_{Y_2})^{rw} \circ (id_{X_1} \ltimes f_2)^{rw}\colon X_1 \times X_2 \to Y_1 \times Y_2$$

The left sequential product models the fact of executing f_1 before f_2, while the right sequential product models the fact of executing f_2 before f_2; in general they return different results and they modify the state in a different way. Sequential products are illustrated in Figure 3 (some indices are omitted).

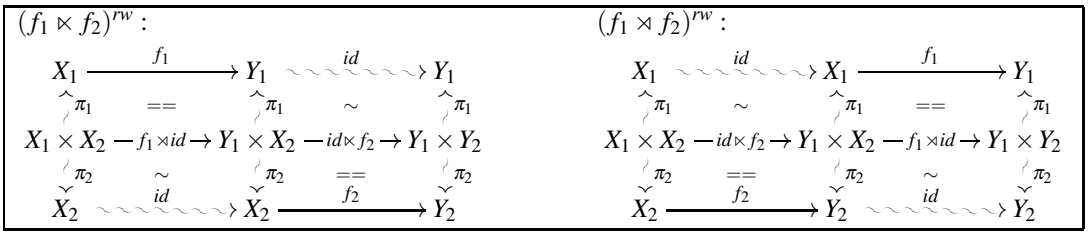

Figure 3: Left and right sequential products

2.4 A property of states

In [8] an equational presentation of states is given, with seven equations. These equations are expressed as decorated equations in [3]. They are the archetype of the properties of which proofs we want to verify. For instance, the fact that modifying a location i and observing the value of another location j can be done in any order is called the *commutation update-lookup* property. This property can be expressed as an equation relating the functions $[\![update_i]\!]$ and $[\![lookup_j]\!]$. For this purpose, let $[\![lookup_j]\!]' : S \times Val_j \times S$ be defined by

$$[\![lookup_j]\!]'(s) = (s, [\![lookup_j]\!](s)) \text{ for each } s \in S\,.$$

Thus, given a state s and a value $a \in Val_i$, assigning a to i and then observing the value of j is performed by the function:

$$[\![lookup_j]\!]' \circ [\![update_i]\!] \colon Val_i \times S \to Val_j \times S\,.$$

Observing the value of j and then assigning a to i also corresponds to a function from $Val_i \times S$ to $Val_j \times S$ built from $[\![update_i]\!]$ and $[\![lookup_j]\!]'$. This function first performs $[\![lookup_j]\!]'(s)$ while keeping a unchanged, then it performs $[\![update_i]\!](s,a)$ while keeping b unchanged (where b denotes the value of j in s which has been returned by $[\![lookup_j]\!](s)$). The first step is $id_{Val_i} \times [\![lookup_j]\!]' \colon Val_i \times S \to Val_i \times (Val_j \times S)$ and the second step is $id_{Val_j} \times [\![update_i]\!] \colon Val_j \times (Val_i \times S) \to Val_j \times S$. An intermediate permutation step is required, it is called $perm_{i,j} \colon Val_i \times (Val_j \times S) \to Val_j \times (Val_i \times S)$ such that $perm_{i,j}(a,(b,s)) = (b,(a,s))$.

Altogether, observing the value of j and then assigning a to i corresponds to the function:

$$(id_{Val_j} \times [\![update_i]\!]) \circ perm_{i,j} \circ (id_{Val_i} \times [\![lookup_j]\!]') \colon Val_i \times S \to Val_j \times S$$

Thus, the commutation update-lookup property means that:

$$[\![lookup_j]\!]' \circ [\![update_i]\!] = (id_{Val_j} \times [\![update_i]\!]) \circ perm_{i,j} \circ (id_{Val_i} \times [\![lookup_j]\!]')$$

According to Section 2.2, this is the interpretation of the following strong equation, which corresponds to the diagram in Figure 4.

$$lookup_j^{ro} \circ update_i^{rw} == \pi_2^{pure} \circ (update_i^{rw} \rtimes id_j^{pure}) \circ (id_i^{pure} \ltimes lookup_j^{ro}) \circ (\pi_1^{-1})^{pure} : V_i \to V_j\,. \tag{1}$$

Remark. Using the right sequential product, the right hand-side of the commutation update-lookup equation can be written as $\pi_2^{pure} \circ (update_i^{rw} \rtimes lookup_j^{ro}) \circ (\pi_1^{-1})^{pure}$.

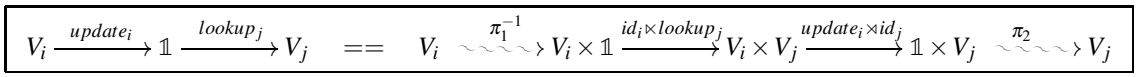

Figure 4: The commutation update-lookup equation

In addition, using the left sequential product, it is easy to check that the left hand-side of this equation can be written as $\pi_2^{pure} \circ (update_i^{rw} \ltimes lookup_j^{ro}) \circ (\pi_1^{-1})^{pure}$. Since $\pi_1^{pure} : V_i \times \mathbb{1} \to V_i$ and $\pi_2^{pure} : \mathbb{1} \times V_j \to V_j$ are invertible, we get a symmetric expression for the equation which corresponds nicely to the description of the commutation update-lookup property as "the fact that modifying a location i and observing the value of another location j can be done in any order":

$$update_i^{rw} \rtimes lookup_j^{ro} == update_i^{rw} \ltimes lookup_j^{ro}$$

3 The Environment in Coq

In this Section we present the core of this paper, namely the implementation in the Coq proof assistant of the rules for reasoning with decorated operations and equations and the proof of the commutation update-lookup property using these rules.

In the preceding section, we shown proofs of propositions involving effects. We now present the construction of a Coq framework enabling one to verify such proofs automatically. This framework has been released as a library and is available in the following web-site: `http://coqeffects.forge.imag.fr`.

In order to construct this framework, we need to define data structures, terms, decorations and basic rules as axioms. Those give rise to derived rules and finally to proofs. This organization is reflected in the library with corresponding Coq modules, as shown on Figure 5.

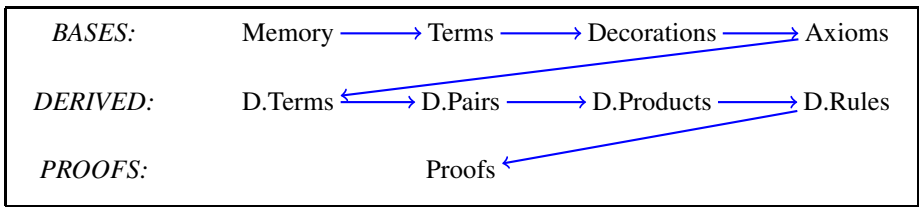

Figure 5: Dependency Chart among Sub-modules

The memory module uses declarations such as *locations* and *location identifiers*. A *location* represents a field on the memory to store and observe data and while *identifiers* correspond to variable names.

Then terms are defined in steps. First we give the definitions of *non-decorated terms*: they constitute the main part of the design with the inclusion of all the required functions. For instance, the `lookup` function which observes the current state is defined

from void ($\mathbb{1}$, the terminal object of the underlying category) to the set of values that could be stored in that specified location.

The next step is to decorate those functions with respect to their manipulation abilities on the state structure. For instance, the `update` function is defined as a *modifier*. the *modifier* status is represented by a *rw* label in the library. All the rules related to decorated functions are stated in the module called *axioms*.

Then, based on the ones already defined, some other terms are derived. For example, the derived `permut` function takes projections as the basis and replaces the orders of input objects in a *categorical product*. Similarly, by using the already defined rules (given in the *axioms* section), some additional rules are derived concerning *categorical pairs*, *products* and *others* pointing the rules constructed over the ones from different sources.

The following subsection we detail the system sub-modules. The order of enumeration gives the dependency among sub-modules as shown in figure 5. For instance, the module *decorations* requires definitions from the *memory* and the *terms* modules. Then, as an example, we give the full proof, in Coq, of the commutation update-lookup property of [8].

3.1 Proof System for States

In this section we give the Coq definitions of our proof system and explore them module by module. Apart from the classical one, a possible alternative order of reading could be to start by the example proof in Section 4 and fiddle backwards to the preceding sections for details on a given rule.

The major ideas in the construction of this Coq framework are:

- All the features of the proof system, that are given in the previous sections 2.2, 2.3, 2.4 and in the appendix A, definitely constitute the basis for the Coq implementation. In brief, we first declare all the terms without decorations, then we decorate them and after all we end up with the rules involving decorated terms. We also confirm that if one removes all the decorations (admitting every operation is `pure`), the proof system keeps smoothly working.

- The constructors `is_comp` and `is_pair` mainly state that two compatible functions could be composed or paired up if they retain the same type of decorations. In order to avoid repeating the same pair and composition constructors for the cases in which input functions are both `pure`, `ro` or `rw`, the variant `k`, representing each decoration type, is used. By this way, mentioned constructors are stated in one line of code such as the others. The constructors `is_pure_ro` and `is_ro_rw` enable us to compose or pair up those compatible functions having different decorations.

- In the proof implementations of some propositions, we use products to specify the evaluation order among functions. In this context, we declare how to construct function pairs, since they are the generic case of products. In some cases, especially when function evaluations are decided to be sequentialized, input functions might required to be permuted. This operation switches the order of sequential function evaluations. In that sense, we derive permuted pairs and so permuted products via the definitions of terms pair and permutation. See subsection 3.6.

- The most challenging part of the design is the proof implementations of the propositions by [8], since they are quite tricky and long. We assert implementations tricky, because to see the main schema (or flowchart) of the proofs at first sight and coding them in Coq with this reasoning is quite difficult. To do so, we first sketch the related diagrams with marked equalities (*strong* or *weak*), then we convert them into some line equations, representing main propositions to be shown, by using fractional notation together with the exploited rules for each step. After all, Coq implementations are done by coding each step took part in the fractional notation. From this aspect, without the fractional correspondences, proofs might be seen a little tough to follow. In order to increase the readability score, we divide those implementations into sub-steps and give relevant explanations for each. See section 4.

- Considering the entire design, we benefit an important aspect provided by Coq environment, *dependent types*. They provide unified formalism in the creation of new data types and allow us to play with all the typing issues for free. More precisely, the type **term** is not a `Type`, but a `Type` → `Type` → `Type`. The domain/codomain information of **term** is embedded into Coq type system, so that we do not need to talk about ill-typed terms. For instance, `pi1 ∘ final` is ill-typed since, `final` is defined from any object X: `Type` to **unit** which is not compatible with `pi1`. Therefore, it cannot be seen as a **term** but anyway, we do not have to state that information explicitly.

3.2 Memory

Let *Loc* be the set of memory locations. A specific Coq `Type`, called `Loc`, is used to represent these locations and is defined as a `Parameter`. This allows it to become globally accessible. For instance, any location `i` ∈ *Loc* is represented in Coq by `i : Loc`, where `Loc` is a global type:

```
Parameter Loc: Type.
```

The values allowed to be stored in any location are defined by another parameter, named `Val`, taking the location identifier as an argument and returning the associated set of values. For instance, for the location identifier `i : Loc`, the possible values allowed to be stored in this location are given as the set `Val i`:

```
Parameter Val: Loc → Type.
```

3.3 Terms

Non-decorated operators, using the monadic equational logic and categorical products, are represented by an inductive (or recursive) Coq data type named **term**. It basically gets two Coq types, that are corresponding either to objects or to mappings in the given categorical structure, and returns a function type. Those function types are the representations of the homomorphisms of the category. We summarize these non-decorated constructions below:

```
Inductive term: Type → Type → Type :=
    | id: ∀ {X: Type}, term X X
    | comp: ∀ {X Y Z: Type}, term X Y → term Y Z → term X Z
    | final: ∀ {X: Type}, term unit X
```

```
| pair: ∀ {X Y Z: Type}, term X Z → term Y Z → term (X×Y) Z
| pi1: ∀ {X Y: Type}, term X (X×Y)
| pi2: ∀ {X Y: Type}, term Y (X×Y)
| lookup: ∀ i: Loc, term (Val i) unit
| update: ∀ i: Loc, term unit (Val i).

Infix "o" := comp (at level 70).
```

For instance, the identity mapping defined from any type to itself is given as **term** X X, taking two types (here two '*X*'s) as input arguments and returning the **term** X X as the output type. It is actually a function mapping the object *X* to itself. The term `comp` composes two given compatible function types and returns another one. The term `pair` represents the categorical product type of two given objects. For instance, if **term** X Z corresponds to a mapping defined from an object Z to another one denoted as X, then `pair` with input types **term** X Z and **term** Y Z, agreeing on domains, returns a new function type of form **term** (X × Y) Z. The terms `pi1` and `pi2` are projections of products while `final` maps any object to the terminal object (the singleton set, denoted by $\mathbb{1}$) of the Cartesian effect category in question. `lookup` takes nothing or null apart from a location identifier and performs a lookup operation for the relevant location. It is mathematically defined from the terminal object of the category to an object denoted by `Val i` (set of values that could be stored in the location identified by i). As the name suggests, the `update` operator updates the value in the specified location, taking any value `a` ∈ `Val i` and returning null. It is defined from an arbitrary object (`Val i`, `Val j`, ...) to the terminal object of the category.

3.4 Decorations

In order to keep the semantics of state close to syntax, all the operations are decorated with respect to their manipulation abilities on the state structure. In Coq, we define another inductive data type, called **kind**, to represent these decorations. Its constructors are `pure` (decorated by 0), `ro` (for read-only and decoration 1) and `rw` (for read-write and decoration 2). It should be recalled that if a function is `pure`, then it could be seen both as `ro` (accessor) and `rw` (modifier), due to the hierarchy rule among decorated functions:

```
Inductive kind := pure | ro | rw.
```

In Coq, we had to define the decorations of terms via the separate inductive data type called **is**. The latter takes term with a kind and returns a `Prop`. In other words, **is** checks whether the given term is allowed to be decorated by the given kind or not. For instance, the term `id` is `pure`, since it cannot use nor modify the state. Therefore it is by definition decorated with 0. This decoration is checked by a constructor `is_id`. To illustrate this, if one (by using `apply` tactic of Coq) asks whether `id` is pure, then the returned result would be have to be `True`. In order to check whether `id` is an accessor or a modifier, the constructors `is_pure_ro` and `is_ro_rw` should be applied beforehand to convert both statements into **is** `pure id`. The incidence of decorations upon the terms is summarized below together with their related rules:

```
Inductive is: kind → ∀ X Y, term X Y → Prop :=
 | is_id: ∀ X, is pure (@id X)                                                    (0-id)            (6)
 | is_comp: ∀ k X Y Z (f: term X Y) (g: term Y Z), is k f → is k g → is k (f ∘ g)  (dec-comp)       (6)
 | is_final: ∀ X, is pure (@final X)                                              (0-final)         (7)
 | is_pair: ∀ k X Y Z (f: term X Z) (g: term Y Z), is k f → is k g → is k (pair f g)  (dec-pair-exists)  (9)
 | is_pi1: ∀ X Y, is pure (@pi1 X Y)                                              (0-proj-1)        (8)
 | is_pi2: ∀ X Y, is pure (@pi2 X Y)                                              (0-proj-2)        (8)
 | is_lookup: ∀ i, is ro (lookup i)                                               (1-lookup)        (10)
 | is_update: ∀ i, is rw (update i)                                               (2-update)        (10)
 | is_pure_ro: ∀ X Y (f: term X Y), is pure f → is ro f                           (0-to-1)          (6)
 | is_ro_rw: ∀ X Y (f: term X Y), is ro f → is rw f                               (1-to-2)          (6)
```

The decorated functions stated above are classified into four different manners:

- terms specific to states effect: `is_lookup` and `is_update`
- categorical terms: `is_id`, `is_comp` and `is_final`
- terms related to categorical products: `is_pair`, `is_pi1` and `is_pi2`.
- term decoration conversions based on the operation hierarchy: `is_pure_ro` and `is_ro_rw`.

The **term** `comp` enables one to compose two compatible functions while the constructor `is_comp` enables one to compose functions and to preserve their common decoration. For instance, if a `ro` function is composed with another `ro`, then the composite function becomes `ro` as well. For the case of the `pair`, the same idea is used. Indeed, the constructor `is_pair` takes two terms agreeing on domains such as **term** Y_1 X, say an `ro`, and **term** Y_2 X, which is `ro` as well. `is_pair` then returns the pair of these terms given as **term** $Y_1 \times Y_2$ X and this is another `ro`. This is realized with both `is_pair` and `is_comp` taking input functions with the same decorations, denoted by `k`, and returning a new function (a pair and a composite, respectively) with the same decoration k. It is also possible to create both compositions and pairs of functions with different decorations via the hierarchy rule stated among decoration types. This hierarchy is build via the last two constructors, `is_pure_ro` and `is_rp_rw`. The constructor `is_pure_ro` indicates the fact that if a term is `pure`, then it can be seen as `ro`. Lastly `is_ro_rw` states that if a term is `ro`, then it can be seen as `rw` as well.

Note that the details of building pairs with different decorations can be found in the derived pairs module (`Pairs.v` in the library).

The same manner could be used to construct compositions of different decorated functions, but due to not being used, they are not specified such as `pairs`.

The terms `final`, `pi1` and `pi2` are all `pure` functions since they do not manipulate the state. `final` forgets its input argument(s) and returns nothing. Although this property could make one think that it generates a sort of side-effect, this is actually not the case. Indeed, it is the only pure function whose co-domain is the terminal object ($\mathbb{1}$) and it is therefore used to simulate the execution of a program: successive, possibly incompatible, functions can then be composed with this intermediate forgetfulness of results.

The `lookup` functions are decorated by 1, as accessors. The constructor `is_lookup` is used to check the validity of the `lookup`' decoration. The different `update` functions are `rw` and decorated by 2. Similarly, the constructor `is_update` is thus used to check the validity of the `update`' decoration.

3.5 Axioms

We can now detail the Coq implementations of the axioms used in the proof constructions. They use the given monadic equational logic and categorical products. The idea is to decorate also the equations. On the one hand, the *weak* equality between parallel morphisms models the fact that those morphisms return the same value but may perform different manipulations of the state. On the other hand, if both the returned results and the state manipulations are identical, then the equality becomes *strong*.

In order to define these decorations of equations in Coq, we again use inductive terms and preserve the naming strategy of Section 2. Both type of equations are equivalence relations and we thus use the Coq definition of relation and import the Relations Morphisms package. This relation statement takes two objects of identical types (**term** X Y in our case) and determines whether those input objects have the same values or not by returning a `Prop`.

Below are given the reserved notations for *strong* and *weak* equalities, respectively.

Reserved Notation " x == y" (at level 80). Reserved Notation "x ∼ y" (at level 80).

We have some number of rules stated w. r. t. *strong* and *weak* equalities. The ones used in the proof given in Section 4 are detailed below:

Inductive **strong**: ∀ X Y, relation (**term** X Y) :=

- The rules `strong_refl`, `strong_sym` and `strong_trans` state that *strong* equality is reflexive, symmetric and transitive, respectively. Obviously, it is an equivalence relation. See (*s*-refl) , (*s*-sym) and (*s*-trans) rules in figure 6.
- Both `id_src` and `id_tgt` state that the composition of any arbitrarily selected function with `id` is itself regardless of the composition order. See (dec-id-src) and (dec-id-tgt) rules in figure 6.
- `strong_subs` demonstrates that strong equality obeys the substitution rule. That means that for a pair of parallel functions that are *strongly* equal, the compositions of the same source compatible function with those functions are still *strongly* equal. `strong_repl` states that for those parallel and *strongly* equal function pairs, their compositions with the same target compatible function are still *strongly* equal. See (*s*-subs) and (*s*-repl) rules in figure 6.
- `ro_weak_to_strong` is the rule saying that all *weakly* equal `ro` terms are also *strongly* equal. Intuitively, from the given *weak* equality, they must have the same results. Now, since they are not modifiers, they cannot modify the state. That means that effect equality requirement is also met. Therefore, they are *strongly* equal. See (ro-*w*-to-*s*) rule in figure 6.

with **weak**: ∀ X Y, relation (**term** X Y) :=

- `pure_weak_repl` demonstrates that *weak* equality obeys the substitution rule stating that for a pair of parallel functions that are *weakly* equal, the compositions of those functions with the same target compatible and `pure` function are still *weakly* equal. See (pure-*w*-repl) rule in figure 6.
- `strong_to_weak` states that *strong* equality could be converted into *weak* one, free of charge. Indeed, the definition of *strong* equality encapsulates the one for *weak* equality. See (*s*-to-*w*) rule in figure 6.

- `axiom_2` states that first updating a location `i` and then implementing an observation to another location `k` is *weakly* equal to the operation which first forgets the value stored in the location `i` and observes location `k`. See (axiom-2) rule in figure 10.

Please note that *weak* equality is an equivalence relation and obeys the substitution rule such as the *strong* one.

3.6 Decorated Terms

Additional to those explained in 3.3, some extra terms are derived via the definitions of already existing ones:

```
Definition inv_pi1 {X Y}: term (X×unit) (X) := pair id unit.
Definition permut {X Y}: term (X×Y) (Y×X) := pair pi2 pi1.
Definition perm_pair {X Y Z} (f: term Y X) (g: term Z X): term (Y×Z)
X
:= permut ∘ pair g f.
Definition prod {X Y X' Y'} (f: term X X') (g: term Y Y'): term (X×Y)
(X'×Y')
:= pair (f ∘ pi1) (g ∘ pi2).
Definition perm_prod {X Y X' Y'} (f: term X X') (g: term Y Y'): term
(X×Y) (X'×Y')
:= perm_pair (f ∘ pi1) (g ∘ pi2).
```

`Val_i` and `Val_i`×𝟙 are isomorphic. Indeed, on the one hand, let us form the left semi-pure pair $h = \langle id_{Val_i}, \langle\rangle_{Val_i}\rangle_r$. As $\langle\rangle_{Val_i}$ is pure, then so is also h. Now, from the definitions of semi-pure products (see Figure 1) the projections yields $\pi_1 \circ h \sim id_{Val_i}$, which is also $\pi_1 \circ h == id_{Val_i}$ since all the terms are pure. On the other hand, $\pi_1 \circ (h \circ \pi_1) == id_{Val_i} \circ \pi_1 == \pi_1 == \pi_1 \circ (id_{Val_i \times \mathbb{1}})$ and $\pi_2 \circ (h \circ \pi_1) == \langle\rangle_{Val_i} \circ \pi_1 \sim \langle\rangle_{Val_i \times \mathbb{1}} \sim \pi_2 == \pi_2 \circ (id_{Val_i \times \mathbb{1}})$. but the latter weak equivalences are strong since all the terms are pure. Therefore $\pi_1 \circ (h \circ \pi_1) == \pi_1 \circ (id_{Val_i \times \mathbb{1}})$ and $\pi_2 \circ (h \circ \pi_1) == \pi_2 \circ (id_{Val_i \times \mathbb{1}})$ so that $h \circ \pi_1 == id_{Val_i \times \mathbb{1}}$. Overall we have that π_1 is invertible and $\pi_1^{-1} = h = \langle id_{Val_i}, \langle\rangle_{Val_i}\rangle_r$ as defined above.

We also have the `permut` term. It takes a product, switches the order of arguments involved in the input product cone and returns the new product: its signature is **term** (Y×X) (X×Y). The **term** `perm_pair f g` is handled via the composition of `pair g f` with `permut`. The definition `prod` is based on the definition of `pair` with a difference that both input functions are taking a product object and returning another one while `perm_prod` is the permuted version of `prod` which is built on `perm_pair`s.

The decorations of `perm_pair`, `prod` and `perm_prod`, depend on the decorations of their input arguments. For instance, a `perm_pair` of two `pure` functions is also `pure` while the `prod` and `perm_prod` of two `rw`s is a `rw`. These properties are provided by `is_perm_pair`, `is_prod` and `is_perm_prod`. More details can be found in the associated module of the library (`Derived_Definitions_Decorations.v`). Note that it is also possible to create `perm_pair`s, `prod`s and `perm_prod`s of functions with different decorations via the hierarchy rule stated among decoration types (`is_pure_ro` and `is_rp_rw`). Existence proofs together with projection rules, can also be found in their respective modules in the library (`Pairs.v` and `Products.v`).

3.7 Decorated Pairs

In this section we present some of the derived rules, related to *pairs* and *projections*. In Section 2.3 we have defined the left semi-pure pair $\langle id_X, f\rangle_l^{rw} : X \to X \times Y$ of the identity id_X^{pure} with a modifier $f^{rw} : X \to Y$. In Coq this construction will be called simply the `pair` of id_X^{pure} and f^{rw}. The right semi-pure pair $\langle f, id_X\rangle_r^{rw} : X \to Y \times X$ of f^{rw} and id_X^{pure} can be obtained as $\langle id_X, f\rangle_l^{rw}$ followed by the permutation $perm_{X,Y} : X \times Y \to Y \times X$, it will be called the `perm_pair` of f^{rw} and id_X^{pure}.

Then, the `pair` and `perm_pair` definitions, together with the hierarchy rules among function classes (`is_pure_ro` and `is_ro_rw`), are used to derive some other rules related to existences and projections.

- `dec_pair_exists_purerw` is the rule that ensures that a `pair` with a `rw` and a `pure` arguments also exists and is `rw` too. `weak_proj_pi1_purerw` is the first projection rule stating that the first result of the pair is equal to the result of its first coefficient function. In our terms it is given as follows: `pi1 ∘ pair f1 f2 ∼ f1`. The given equality is *weak* since its left hand side is `rw`, while its right hand side is `pure`. `strong_proj_pi2_purerw` is the second projection rule of the semi-pure pair. It states that the second result of the pair and its effect are equal to the result and effect of its second coefficient function. In our terms it is given as follows: `pi2 ∘ pair f1 f2 == f2`.

- `dec_perm_pair_exists_rwpure` is similar with pure and modifier inverted.

- `dec_pair_exists_purepure` is similar but with both coefficient functions `pure`. Thus it must be pure by itself and its projections must be strongly equal to its coefficient functions. These properties are stated via `strong_proj_pi1_purepure` (`pi1 ∘ pair f1 f2 == f1`) and `pure`. `strong_proj_pi2_purepure` (`pi2 ∘ pair f1 f2 == f2`).

More details can be found in the `Pairs.v` source file.

3.8 Decorated Products

Semi-pure products are actually specific types of semi-pure pairs, as explained in Section 2.3. In the same way in Coq, the `pair` and `perm_pair` definitions give rise to the `prod` and `perm_prod` ones.

- `dec_prod_exists_purerw` ensures that a `prod` with a `pure` and a `rw` arguments exists and is `rw`.

  `weak_proj_pi1_purerw_rect` is the first projection rule and states that `pi1 ∘ (prod f g) ∼ f ∘ pi1`. `strong_proj_pi2_purerw_rect` is the second projection rule and assures that `pi2 ∘ (prod f g) == f ∘ pi2`.

- Similarly, the rules `dec_perm_prod_exists_rwpure`, `strong_perm_proj_pi1_rwpure_rect` (`pi1 ∘ (perm_prod f g) == f ∘ pi1`) and `weak_perm_proj_pi2_rwpure_rect` (`pi2 ∘ (perm_prod f g) ∼ g ∘ pi2`) relate to permuted products.

For further explanation of each derivation with Coq implementation, refer to the `Products.v` source file.

3.9 Derived Rules

We detail here some other derived, used in the example proof given next.

- `weak_refl` describes the reflexivity property of the weak equality.
- `comp_final_unique` ensures that two parallel `rw` functions (say f and g) are the same (strongly equal) if they return the same result `pi1 ∘ f ∼ pi1 ∘ g` and have the same effect `pi2 ∘ f == pi2 ∘ g`.
- Two pure functions have the same co-domain $\mathbb{1}$ must be strongly equal (no result and state unchanged). Therefore `E_0_3` extends this to a composed function $f \circ g$, for two pure compatible functions f and g, and another function h, provided that g and h have $\mathbb{1}$ as co-domain. In the same parallel, `E_1_4` states that the composition of any `ro` function h: $\mathbb{1} \to X$, with `final` is strongly equal to the `id` function on $\mathbb{1}$. Indeed, the bioth have no result and do not modify the state since h is not a modifier.

More similar derived rules can be found in the `Derived.v` source file.

4 Implementation of a Proof: Commutation update-lookup

We now have all the ingredients required to prove the commutation update-lookup property of Figure 4: it states that the order of operations between updating a location and retrieving the value at another location does not matter. The formal statement is given in Equation (1): `lookup j ∘ update i == pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1`. The value intended to be stored into the location `i` is an element of `Val_i` set while the lookup operation to the location `j` takes nothing (apart from `j`), and returns a value read from the set `Val_j`. If the order of operations is reversed, then the element of `Val_i` has to be preserved while the other location is examined. Thus we need to form a pair with the identity and create a product `Val_i × 𝟙`, via `inv_pi1`. Similarly, the value recovered by the lookup operation has to be preserved and returned after the update operation. Then a pair with the identity is also created with update and a last projection is used to separate their results. The full Coq proof thus uses the following steps:

1. assume `i, j:Loc`
2. `lookup j ∘ update i == pi2 ∘ (perm_prod (update i) id)`
3. `∘ (prod id (lookup j)) ∘ inv_pi1` by `comp_final_unique`
4. step 1
5. `final ∘ lookup j ∘ update i == final`
6. `∘ pi2 ∘ (perm_prod (update i) id)` by `strong_sym`
7. `∘ (prod id (lookup j)) ∘ inv_pi1`
8. `final ∘ pi2 ∘ (perm_prod (update i) id)`
9. `∘ (prod id (lookup j)) ∘ inv_pi1`

```
10.     == final ∘ lookup j ∘ update i                                  by strong_trans
11.   substep 1.1
12.     final ∘ pi2 ∘ (perm_prod (update i) id)
13.     ∘ (prod id (lookup j)) ∘ inv_pi1
14.     == pi1 ∘ (perm_prod (update i) id)
15.     ∘ (prod id (lookup j)) ∘ inv_pi1                                by E_0_3
16.   substep 1.2
17.     pi1 ∘ (perm_prod (update i) id)
18.     ∘ (prod id (lookup j)) ∘ inv_pi1
19.     == update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1              by strong_perm_proj
20.   substep 1.3
21.     update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1
22.     == update i ∘ pi1 ∘ inv_pi1                                     by strong_proj
23.   substep 1.4
24.     update i ∘ pi1 ∘ inv_pi1 == update i ∘ id                       by id_tgt
25.   substep 1.5
26.     final ∘ lookup j ∘ update i == update i ∘ id                    by E_1_4
27. step 2
28.   lookup j ∘ update i ∼ pi2 ∘ (perm_prod (update i) id)
29.   ∘ (prod id (lookup j)) ∘ inv_pi1                                  by weak_trans
30.   substep 2.1
31.     lookup j ∘ update i ∼ lookup j ∘ final                          by axiom_2
32.   substep 2.2
33.     lookup j ∘ final ∼ lookup j ∘ pi2 ∘ inv_pi1                     see § 3.7
34.   substep 2.3
35.     pi2 ∘ (perm_prod (update i) id)
36.     ∘ (prod id (lookup j)) ∘ inv_pi1
37.     ∼ pi2 ∘ (prod id (lookup j)) ∘ inv_pi1                          by strong_perm_proj
38.   substep 2.4
39.     pi2 ∘ (prod id (lookup j)) ∘ inv_pi1
40.     ∼ lookup j ∘ pi2 ∘ inv_pi1                                      by strong_proj
41. lookup j ∘ update i == pi2 ∘ (perm_prod (update i) id)
42. ∘ (prod id (lookup j)) ∘ inv_pi1
```

To prove such a proposition, the comp_final_unique rule is applied first and results in two sub-goals to be proven: `final ∘ lookup j ∘ update i == final ∘ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1` (to check if both hand sides have the same effect or not) and `lookup j ∘ update i ∼ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1` (to see whether they return the same result or not). Proofs of those sub-goals are given in step

1 and step 2, respectively.

Step 1. `final ∘ lookup j ∘ update i == final ∘ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1`:

(1.1) The left hand side of the equation, `final ∘ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1`, (after the `strong_sym` rule application) is reduced into: `pi1 ∘ (perm_prod (update i) id) ∘ (prod (id (Val i)) (lookup j)) ∘ inv_pi1`. The base point is an application of `E_0_3`, stating that `final ∘ pi2 == pi1` and followed by `strong_subs` applied to `(perm_prod (update i) id)`, `(prod id (lookup j))` and `inv_pi1`.

(1.2) In the second sub-step, `strong_perm_proj_pi1_rwpure_rect` rule is applied to indicate the strong equality between `pi1 ∘ perm_prod (update i) id` and `update i ∘ pi1`. After the applications of `strong_subs` with arguments `prod id (lookup j)` and `inv_pi1`, we get: `pi1 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1 == update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1`. Therefore, the left hand side of the equation can now be stated as: `update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1`.

(1.3) Then, the third sub-step starts with the application of `weak_proj_pi1_purerw_rect` rule in order to express the following weak equality: `pi1 ∘ prod id (lookup j) ∼ id ∘ pi1`. The next step is converting the existing weak equality into a strong one by the application of `ro_weak_to_strong`, since none of the components are modifiers. Therefore we get: `pi1 ∘ prod id (lookup j) == id ∘ pi1`. Now, using `id_tgt`, we remove `id` from the right hand side. The subsequent applications of `strong_subs` with arguments `inv_pi1` and `strong_repl` enables us to relate `update i ∘ pi1 ∘ (prod id (lookup j)) ∘ inv_pi1` with `update i ∘ pi1 ∘ inv_pi1` via a strong equality.

(1.4) In this sub-step, `update i ∘ pi1 ∘ inv_pi1` is simplified into `update i ∘ id`. To do so, we start with `strong_proj_pi1_purepure` so that `pi1 ∘ pair id final == id`, where `pair id final` defines `inv_pi1=pair id final`. Then, the application of `strong_repl` to `update i` provides: `update i ∘ pi1 ∘ pair id final == update i ∘ id`.

(1.5) In the last sub-step, the right hand side of the equation, `final ∘ lookup j ∘ update i`, is reduced into `update i ∘ id`. To do so, we use `E_1_4` which states that `final ∘ lookup j == id`. Then, using `strong_subs` on `update i`, we get: `final ∘ lookup j ∘ update i == id ∘ update i`. By using `id_tgt` again we remove `id` on the right hand side and `id_src` rewrites `final ∘ lookup j ∘ update i` as `update i ∘ id`.

At the end of the third step, the left hand side of the equation is reduced into the following form: `update i ∘ id` via a `strong` equality. Thus, in the fourth step, it was sufficient to show `final ∘ lookup j ∘ update i == update i ∘ id` to prove `final ∘ pi2 ∘ (perm_prod (update i) id)) ∘ (prod id (lookup j)) ∘ inv_pi1 == final ∘ lookup j ∘ update i`. This shows that both sides have the same effect on the state structure.

Step 2. We now turn to the second step of the proof, namely: `lookup j ∘ update i ∼ pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1`.

The results returned by both input composed functions are examined. Indeed, from step 1 we know that they have the same effect and thus if they also return the same results, then they we will be strongly equivalent.

(2.1) Therefore, the first sub-step starts with the conversion of the left hand side of the equation, `lookup j ∘ update i`, into `lookup j ∘ final` via a weak equality. This is done by the application of the `axiom_2` stating that `lookup j ∘ update i` $\sim$ `lookup j ∘ final` for `j` $\neq$ `i`.

(2.2) The second sub-step starts with the application of `strong_proj_pi2_purepure` which states `pi2 ∘ (pair id final) == final` still with `(pair id final) = inv_pi1`. Then, via the applications of `strong_repl`, with argument `lookup j`, `strong_to_weak` and `strong_sym`, we get: `lookup j ∘ final` $\sim$ `lookup j ∘ pi2 ∘ inv_pi1`.

(2.3) In the third sub-step, the right hand side of the equation, `pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1`, is simplified by `weak` equality: we start with the application of `weak_perm_proj_pi2_rwpure_rect` since `pi2 ∘ (perm_prod (update i) id)` $\sim$ `id ∘ pi2`. Then, we once again use `id_tgt` to remove the identity and the applications of `weak_subs` with arguments `prod id (lookup j)` and `inv_pi1` yields the following equation: `pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1` $\sim$ `pi2 ∘ (prod id (lookup j)) ∘ inv_pi1`.

(2.4) In the last sub-step, `pi2 ∘(prod id (lookup j)) ∘ inv_pi1` is reduced into `lookup j ∘ pi2 ∘ inv_pi1` via a weak equality using `strong_proj_pi2_purerw_r` so that `pi2 ∘ (prod id (lookup j)) == lookup j ∘ pi2`. Then, `strong_repl` is applied with argument `inv_pi1`. Finally, the `strong_to_weak` rule is used to convert the `strong` equality into a `weak` one.

Both hand side operations return the same results so that the statement `lookup j ∘ update i` $\sim$ `pi2 ∘ (perm_prod (update i) id) ∘ (prod id (lookup j)) ∘ inv_pi1` is proven.

Merging the two steps (same effect and same result) yields the proposition that both sides are strongly equal. The full Coq development can be found in the library in the source file `Proofs.v`. □

5 Conclusion

In this paper, we introduce a framework implemented in Coq proof assistant. The main goal of the implementation is enabling programmers to verify properties of programs with the global states effect. We present those properties close to syntax meaning that the state structure itself is not mentioned in the verification progress. Instead, it is represented by the term decorations that are used to declare program properties. We also remark that such type of a framework should definitely serve the proof implementations of the propositions by [8]. To prove them, we benefit the mathematical structure proposed by [3] in which the effect of any operation (function) is defined as the distance from being pure and denoted as $\langle\rangle_Y \circ$ `f` where `f: X` $\rightarrow$ `Y`. Therefore, for the specific case of `states` effect, to state the *strong* equality between any parallel morphisms `f, g: X` $\rightarrow$ `Y`, we first check whether they have the same effect over the existence of following equation: $\langle\rangle_Y \circ$ `f` $==$ $\langle\rangle_Y \circ$ `g` and then monitor if they return

the same result depending on the existence of the equation `f` $\sim$ `g`. To illustrate, section 4 gives one of the proofs of the mentioned propositions in detail according to the corresponding steps stated in Coq environment.

By using the way stated in this paper, the next step is to build another framework (also in Coq) which lets programmers to prove properties of programs including `exceptions`. This work is planned to be followed by the study of composing `states` effect with `exceptions`. In other words, the idea is to end up with one general framework in which properties of programs with only `states` effect or only `exceptions` effect or both at the same time are enabled to be proven.

References

[1] B. Ahrens and J. Zsido. Initial semantics for higher-order typed syntax. *Journal of Formalized Reasoning*, 4(1), 2011. http://arxiv.org/abs/1012.1010.

[2] Y. Bertot. *From Semantics to Computer Science, essays in Honour of Gilles Kahn*, chapter Theorem proving support in programming language semantics, pages 337–361. Cambridge University Press, 2009. http://hal.inria.fr/inria-00160309/.

[3] J.-G. Dumas, D. Duval, L. Fousse, and J.-C. Reynaud. Decorated proofs for computational effects: States. In U. Golas and T. Soboll, editors, *ACCAT*, volume 93 of *EPTCS*, pages 45–59, 2012. http://hal.archives-ouvertes.fr/hal-00650269.

[4] J.-G. Dumas, D. Duval, and J.-C. Reynaud. Cartesian effect categories are freyd-categories. *J. Symb. Comput.*, 46(3):272–293, 2011. http://hal.archives-ouvertes.fr/hal-00369328.

[5] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[6] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *POPL*, pages 47–57. ACM Press, 1988.

[7] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[8] G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[9] J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical. Structures in Comp. Sci.*, 7(5):453–468, Oct. 1997.

[10] G. Stewart. Computational verification of network programs in coq. In *Proceedings of Certified Programs and Proofs (CPP 2013), Melbourne, Australia*, Dec. 2013. http://www.cs.princeton.edu/~jsseven/papers/netcorewp.

[11] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003.

A Decorated rules

In order to prove mentioned propositions by Plotkin et al [8], we introduce some number of rules based on three different subjects stated as follows: *monadic equational logic*, *categorical products* and *observational products*.

A.0.1 Rules Related to Monadic Equational Logic

Monadic equational logic is a category C in which the axioms constructing the category in question exist up to two equivalence relations, denoted by "==" and "$\sim$" in this paper. In more precise terms, monadic equational logic is a directed graph whose vertices are objects and edges are morphisms together with the satisfaction of the rules given in Table 6 where d-type decorations could be either of *pure*, *ro* or *rw*.

$$\text{(0-id)}\ \frac{X}{id_X^{pure} : X \to X} \qquad \text{(dec-comp)}\ \frac{f^{(d)} : X \to Y \quad g^{(d)} : Y \to Z}{(g \circ f)^{(d)} : X \to Z} \qquad \text{(0-to-1)}\ \frac{f^{pure}}{f^{ro}} \qquad \text{(1-to-2)}\ \frac{f^{ro}}{f^{rw}}$$

$$\text{(}s\text{-refl)}\ \frac{f^{rw}}{f == f} \qquad \text{(}s\text{-sym)}\ \frac{f^{rw} == g^{rw}}{g == f} \qquad \text{(}s\text{-trans)}\ \frac{f^{rw} == g^{rw} \quad g^{rw} == h^{rw}}{f == h}$$

$$\text{(dec-assoc)}\ \frac{f^{rw} : X \to Y \quad g^{rw} : Y \to Z \quad h^{rw} : Z \to W}{h \circ (g \circ f) == (h \circ g) \circ f} \qquad \text{(dec-id-src)}\ \frac{f^{rw} : X \to Y}{f \circ id_X == f} \qquad \text{(dec-id-tgt)}\ \frac{f^{rw} : X \to}{id_Y \circ f ==}$$

$$\text{(}s\text{-subs)}\ \frac{f^{rw} : X \to Y \quad g_1^{rw} == g_2^{rw} : Y \to Z}{g_1 \circ f == g_2 \circ f : X \to Z} \qquad \text{(}s\text{-repl)}\ \frac{f_1^{rw} == f_2^{rw} : X \to Y \quad g^{rw} : Y \to Z}{g \circ f_1 == g \circ f_2 : X \to Z}$$

$$\text{(ro-}w\text{-to-}s\text{)}\ \frac{f^{ro} \sim g^{ro}}{f == g} \qquad \text{(}s\text{-to-}w\text{)}\ \frac{f^{rw} == g^{rw}}{f \sim g} \qquad \text{(}w\text{-sym)}\ \frac{f^{rw} \sim g^{rw}}{g \sim f} \qquad \text{(}w\text{-trans)}\ \frac{f^{rw} \sim g^{rw} \quad g^{rw} \sim h^{rw}}{f \sim h}$$

$$\text{(}w\text{-subs)}\ \frac{f^{rw} : X \to Y \quad g_1^{rw} \sim g_2^{rw} : Y \to Z}{g_1 \circ f \sim g_2 \circ f : X \to Z} \qquad \text{(pure-}w\text{-repl)}\ \frac{f_1^{rw} \sim f_2^{rw} : X \to Y \quad g^{pure} : Y \to Z}{g \circ f_1 \sim g \circ f_2 : X \to Z}$$

Figure 6: Rules of the decorated monadic equational logic for states

For instance, the rule (0-to-1) is stating that if a function is pure, then it could be treated as an accessor while (1-to-2) rules an accessor to be counted as a modifier.

A.0.2 Rules Related to Categorical Products

Due to the background mathematical structure, which is a Cartesian Category, rules concerning sequential products (compositions of semi-pure ones) have to be stated by definition. The rules related to existence and projections of binary products, existence of empty products, existence and projections of left morphism pairs are given in the following tables 7, 8 and 9, respectively in which d-type decorations could be either of *pure*, *ro* or *rw*. The existence and projections of right morphism pairs, left and right morphism products together with the rules stating the existences of forgetfulness, permutation and isomorphism are the ones could be derived and not stated in this paper.

$$\text{(unit-exists)}\ \frac{}{\mathbb{1}} \qquad \text{(0-final)}\ \frac{X}{\langle\rangle_X^{pure} : X \to \mathbb{1}} \qquad \text{(}w\text{-final-unique)}\ \frac{f^{rw},\ g^{rw} : X \to \mathbb{1}}{f \sim g}$$

$$\text{(dec-comp-final-unique)}\ \frac{f^{rw},\ g^{rw} : X \to Y \quad \langle\rangle_Y^{pure} \circ f^{rw} == \langle\rangle_Y^{pure} \circ g^{rw} \quad f^{rw} \sim g^{rw}}{f == g}$$

Figure 7: Rules of the decorated empty products for states

One of the most important rules given in this context is (dec-comp-final-unique) (in figure 7) which checks both the state effects of the input parallel functions (`f` and `g`) and their results. If they have the same effect $\langle\rangle \circ$ `f` == $\langle\rangle \circ$ `g` and the same returned results `f` $\sim$ `g`, then they are called *strongly equal* and denoted as follows: `f == g`.

The (w-pair-unique) (in figure 9) is another important rule for parallel functions (`f` and `g`) returning more than one arguments (categorical products are used for the representation issues). It mainly checks the returned results of both functions. `pi_1` $\circ$ `f` $\sim$ `pi_1` $\circ$ `g` stands for the first argument comparison and `pi_2` $\circ$ `f` $\sim$ `pi_2` $\circ$ `g` for the second one. If both are the same, then it is obvious to state that those input functions do return the same results after evaluations, shown as follows: `f` $\sim$ `g`. There is nothing more to say.

$$\text{(prod-exists)}\ \frac{X_1 \quad X_2}{X_1 \times X_2} \qquad \text{(0-proj-1)}\ \frac{X_1 \quad X_2}{\pi_{X_1,X_2,1}^{pure} : X_1 \times X_2 \to X_1} \qquad \text{(0-proj-2)}\ \frac{X_1 \quad X_2}{\pi_{X_1,X_2,2}^{pure} : X_1 \times X_2 \to X_2}$$

Figure 8: Rules of the decorated binary products for states: Existence

$$\text{(dec-pair-exists)}\ \frac{f_1^{(d)} : X \to Y_1 \quad f_2^{(d)} : X \to Y_2}{< f_1, f_2 >^{(d)} : X \to Y_1 \times Y_2}$$

$$\text{(dec-pair-proj-1)}\ \frac{f_1^{ro} : X \to Y_1 \quad f_2^{rw} : X \to Y_2}{\pi_{Y_1,Y_2,1} \circ < f_1, f_2 > \sim f_1} \qquad \text{(dec-pair-proj-2)}\ \frac{f_1^{ro} : X \to Y_1 \quad f_2^{rw} : X \to Y_2}{\pi_{Y_1,Y_2,2} \circ < f_1, f_2 > == f_2}$$

$$\text{(}w\text{-pair-unique)}\ \frac{f^{rw}, g^{rw} : X \to Y_1 \times Y_2 \quad \pi_{Y_1,Y_2,1}^{pure} \circ f^{rw} \sim \pi_{Y_1,Y_2,1} \circ g^{rw} \quad \pi_{Y_1,Y_2,2}^{pure} \circ f^{rw} \sim \pi_{Y_1,Y_2,2}^{pure} \circ g^{rw}}{f \sim g}$$

Figure 9: Rules of the decorated pairs for states: Existence & Unicity

A.0.3 Rules of Observational Products

As the name suggests, observational products let us define the types of equalities between functions or function compositions by arranging observations to the memory locations which might be used or modified.

Let Loc be the set of locations. Then, rules for each location $i, k \in Loc$ where $i \neq k$ are given as follows: where V_i represents the set of possible values that could be stored

$$\frac{}{V_i} \quad \text{(1-lookup)}\ \frac{}{lookup_i^{ro} : \mathbb{1} \to V_i} \quad \text{(2-update)}\ \frac{}{update_i^{rw} : V_i \to \mathbb{1}}$$

$$\text{(axiom-1)}\ \frac{}{lookup_i \circ update_i \sim id_i} \quad \text{(axiom-2)}\ \frac{}{lookup_i \circ update_k \sim lookup_i \circ \langle\rangle_k}$$

$$\text{(dec-local-to-global)}\ \frac{f^{rw}, g^{rw} : X \to \mathbb{1} \quad \textit{for each location } k,\ lookup_k^{ro} \circ f^{rw} \sim lookup_k^{ro} \circ g^{rw}}{f == g}$$

Figure 10: Rule of the decorated observational products for states

in the location pointed by i, while $lookup_i$ and $update_i$ correspond to `lookup` and `update` operations that are performed on the same location.

The rule (axiom-1) states that by updating a specific location pointed by `i` $\in$ *Loc* and then reading the stored value, one gets the input value of the latest update operation. If this value is passed to an identity function, it gets returned as it is. Therefore, `lookup i` $\circ$ `update i` $\sim$ `id_i`. Because, left hand side composition returns the same result with right hand side function but different state effects: `lookup i` $\circ$ `update i` is a modifier while `id_i` is pure.

The rule (axiom-2) indicates that by updating a location identified by `k` $\in$ *Loc* and then observing another location identified by `i` $\in$ *Loc*, one gets the stored value in the location pointed by `i`. On the other hand, forgetting the value stored in the location pointed by `k` and reading the one located in `i`, one gets the value stored in `i`. Therefore, `lookup_i` $\circ$ `update_k` $\sim$ `lookup_i` $\circ$ $\langle\rangle$`_k`. Both hand sides return the same result but left hand side composition is a modifier while right hand side one is an accessor.

(dec-local-to-global) states that being able to define equivalences between given two parallel morphisms, one of the strategies to be followed is that all of the memory locations are observed before and after the applications of both input functions. They are said to be *strongly equal* if all the values stored in the every location are the same. Since, too many local observations yield in a global one, meaning that both of the input functions have the same effect on the state structure.