





# Machine-Assisted Proofs for Institutions in Coq

Conor Reynolds<sup>(✉)</sup>  and Rosemary Monahan 

Maynooth University, Maynooth, Ireland  
{conor.reynolds,rosemary.monahan}@mu.ie

**Abstract.** The theory of institutions provides an abstract mathematical framework for specifying logical systems and their semantic relationships. Institutions are based on category theory and have deep roots in a well-developed branch of algebraic specification. However, there are no machine-assisted proofs of correctness for institution-theoretic constructions—chiefly satisfaction conditions for institutions and their (co)morphisms—making them difficult to incorporate into mainstream formal methods. This paper therefore provides the details of our approach to formalizing a fragment of the theory of institutions in the Coq proof assistant. We instantiate this framework with the institutions *FOPEQ* for first-order predicate logic and *EVT* for the Event-B specification language, both of which will serve as an illustration and evaluation of the overall approach.

## 1 Introduction

The theory of institutions dates to Joseph Goguen and Rod M. Burstall’s 1984 paper [7] and the subsequent more detailed analysis in 1992 [8]. An institution is a mathematical realisation of the notion of “logical system” which does not commit to any single concrete system. The key insight is that many general results about logical systems do not depend in any interesting way on the details of that system.

In her PhD thesis [6], Marie Farrell uses the theory of institutions to provide a semantics for the Event-B formal modelling method with an eye to addressing some drawbacks of the Event-B language—namely the lack of standardised modularisation constructs. *EVT* was shown by Farrell [6], on paper, to support such constructs.

Indeed, the theory of institutions has been applied to a wide variety of languages and formal methods; CLEAR [2], CSP [15], and UML [10] have been given an institution-theoretic semantics, to name but a few. The HETS tool for heterogeneous specification [12] has the largest single repository of such institutions and their logical relationships, represented mainly by institution morphisms and comorphisms; but as far as we know there are no machine-checked

---

Funded by the Irish Research Council (GOIPG/2019/4529).

© Springer Nature Switzerland AG 2022

Y. Aït-Ameur and F. Crăciun (Eds.): TASE 2022, LNCS 13299, pp. 180–196, 2022.

[https://doi.org/10.1007/978-3-031-10363-6\\_13](https://doi.org/10.1007/978-3-031-10363-6_13)

proofs that these constructions are correct. Many of the requirements—checking that categories are really categories, that functors are really functors, as well as satisfaction conditions for institutions and for the (co)morphisms that relate them—amount in some parts to simple bookkeeping, and in other parts to more novel and interesting results.

We hence provide here a framework in the Coq proof assistant [5] for interactive machine-assisted proofs for institutions and an instantiation of this framework to two institutions: the institution *FOPEQ* for first-order predicate logic and the institution *EVT* for Event-B. Coq has two properties desirable for this work. First, it is based on a dependent type theory called the calculus of inductive constructions (CIC) which makes the representation of mathematical objects and the subtle constraints that they impose on one another easier than in a system without dependent types. Second, it is an interactive proof assistant rather than an automated proof assistant. The user can design automated tactics that can discharge many simple goals, but crucially Coq allows the user to step in and spell out the proofs in detail if necessary. Our framework is available on GitHub at <https://github.com/ConorReynolds/coq-institutions>.

We build directly on the work done by Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano [9] formalizing multi-sorted universal algebra in Agda. We also note some other work in this direction in Coq by Venanzio Capretta [3], and by Gianluca Amato, Marco Maggesi and Maurizio Parton and Cosimo Perini Brogi [1] which makes use of homotopy type theory—but none go quite as far as defining institutions or instantiating first-order logic at the time of this writing. This is the first such formalization of which we are aware.

We will begin by laying the basic mathematical groundwork for institution theory, multi-sorted universal algebra, and first-order predicate logic, before explaining how these concepts are defined in our Coq developments. First-order logic is an extremely central institution, on which many others build (including *EVT*) and provides an appropriate first example. We then provide the same treatment for *EVT* as a further case study, and to provide a concrete example of one institution building on another.

## 2 Mathematical Background

Institutions are based on category theory. A *category* consists of a collection of objects, and a collection of arrows or morphisms between those objects, subject to some straightforward laws. A *functor* is a map between categories which preserves the categorical structure—more precisely, it preserves identity morphisms and composition of morphisms. Definitions for these concepts can be found in Emily Riehl’s freely available *Category Theory in Context* [14]. We only require very light familiarity with categories and functors for this paper.

**Definition 1.** An institution [7] consists of

- a category  $\text{Sig}$  of signatures;
- a sentence functor  $\text{Sen} : \text{Sig} \rightarrow \text{Set}$ ;

- a model functor  $\text{Mod} : \text{Sig}^{\text{op}} \rightarrow \text{Cat}$ ; and
- a semantic entailment relation  $\models_{\Sigma} \subseteq |\text{Mod}(\Sigma)| \times \text{Sen}(\Sigma)$  for each  $\Sigma \in \text{Sig}$ ,

such that for any signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ , any sentence  $\phi \in \text{Sen}(\Sigma)$ , and any model  $M' \in \text{Mod}(\Sigma')$ , the satisfaction condition holds:

$$M' \models_{\Sigma'} \text{Sen}(\sigma)(\phi) \quad \text{iff} \quad \text{Mod}(\sigma)(M') \models_{\Sigma} \phi$$

ensuring that a change in signature induces a consistent change in the satisfaction of sentences by models.

The signatures contain non-logical symbols: data types, constants, functions, and so on. The sentence functor explains how to build sentences over the non-logical symbols. The model functor explains how to interpret the symbols in any given signature. The semantic entailment relation explains how to decide if a given sentence is true or false in a given model. The requirement that the signatures form a category, and that the sentence and model constructors are functors, is due to the central concept of a *signature morphism*, a mapping between signatures—a “change in notation”. If the sentence construction is a functor then we can be sure that signature translations preserve the sentence structure.

The satisfaction condition explains how the components should interact with one another, and in particular how they behave under a change in signature. Without such a condition, the semantic entailment relation  $\models$  could behave just as expected on one signature, but behave utterly erratically on another. But we expect the entailment relation  $\models$  to change only so much with a change in signature. The satisfaction condition ensures that satisfaction of sentences by models is consistent under a change in signature.

## 2.1 First-Order Predicate Logic

We provide a brief account of multi-sorted universal algebra and first-order predicate logic, in preparation for a formal encoding in Coq (Sect. 4); see Sannella and Tarlecki’s *Foundations of Algebraic Specification* [17] for details.

**Definition 2.** An  $S$ -indexed set is a family of sets  $X = (X_s)_{s \in S}$ .

**Definition 3.** A signature is a 3-tuple  $\langle S, \mathcal{F}, \mathcal{P} \rangle$  where  $S$  is a set of sorts,  $\mathcal{F}$  is a  $(\text{List}(S) \times S)$ -indexed set of function symbols, and  $\mathcal{P}$  is a  $\text{List}(S)$ -indexed set of predicate symbols.

Here  $\text{List}(A)$  is just as expected: the set of all finite sequences of elements from  $A$ . The idea is that a symbol  $F \in \mathcal{F}_{w,s}$  has arity  $w$  and result sort  $s$ , and a predicate symbol  $P \in \mathcal{P}_w$  has arity  $w$  and no result sort (since it represents a predicate). If the signature is clear from context, we instead write  $F : \prod_i w_i \rightarrow s$  for function symbols, and  $P : \prod_i w_i \rightarrow \text{Prop}$  for predicate symbols. If a function symbol  $C$  has arity  $\text{nil}$  and result sort  $s$ , then it is called a constant symbol and we denote it  $C : s$ .

As a running example, let `stackSig` be a signature consisting of the symbols required to describe a stack. It has two sorts `elem` and `stack`; some function symbols `empty` : `stack`, `push` : `elem` × `stack` → `stack`, and `pop` : `stack` → `stack`; and a predicate symbol `isEmpty` : `stack` → `Prop`.

**Definition 4.** Let  $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$  and  $\Sigma' = \langle S', \mathcal{F}', \mathcal{P}' \rangle$  be two signatures. A signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  consists of a function  $\sigma_{\text{sorts}} : S \rightarrow S'$ , which will usually be written  $\sigma$ , as well as a pair of functions

$$\begin{aligned} \sigma_{\text{funcs}} &: \prod_{w,s} \mathcal{F}_{w,s} \rightarrow \mathcal{F}'_{\sigma(w),\sigma(s)} \\ \sigma_{\text{preds}} &: \prod_w \mathcal{P}_w \rightarrow \mathcal{P}'_{\sigma(w)} \end{aligned}$$

respectively mapping sorts, function symbols, and predicate symbols, in such a way that the sorts are translated consistently with  $\sigma_{\text{sorts}}$ . We define  $\sigma(w)$  as the action of  $\sigma_{\text{sorts}}$  on each of the sorts in  $w$ .

**Definition 5.** An algebra  $A$  for a signature  $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$  consists of three functions  $\langle A_{\text{sorts}}, A_{\text{funcs}}, A_{\text{preds}} \rangle$ , all of which we denote by  $A$ , each respectively interpreting the sorts, function symbols, and predicate symbols as sets, functions, and predicates:

- for any sort  $s \in S$ ,  $A(s)$  is a set, which we typically denote  $A_s$ ;
- for any  $F \in \mathcal{F}_{w,s}$ , we have  $A(F) : A_{w_1} \times \cdots \times A_{w_n} \rightarrow A_s$ ; and
- for any  $P \in \mathcal{P}_w$ , we have  $A(P) \subseteq A_{w_1} \times \cdots \times A_{w_n}$ .

Algebras give meaning to the symbols in a signature. Consider again our running example `stackSig`; we could interpret the sort `elem` as the set  $\mathbb{N}$  of natural numbers, and the sort `stack` as the set  $\text{List}(\mathbb{N})$  of lists of natural numbers; the function symbols `empty`, `push`, and `pop` as  $\text{nil} \in \text{List}(\mathbb{N})$ ,  $\text{cons} : \mathbb{N} \times \text{List}(\mathbb{N}) \rightarrow \text{List}(\mathbb{N})$ , and  $\text{tail} : \text{List}(\mathbb{N}) \rightarrow \text{List}(\mathbb{N})$ , respectively; and the predicate symbol `isEmpty` as the predicate  $\{s \mid s = \text{nil}\} \subseteq \text{List}(\mathbb{N})$ . We are by no means bound to this interpretation, of course.

**Definition 6.** Let  $\Sigma$  and  $\Sigma'$  be signatures, let  $\sigma : \Sigma \rightarrow \Sigma'$  be a signature morphism, and let  $A'$  be a  $\Sigma'$ -algebra. The reduct algebra  $A'|_{\sigma}$  is a  $\Sigma$ -algebra defined at each component of the algebra to be  $A' \circ \sigma$ .

Algebras are best thought of (loosely) as functions providing a concrete denotation for the symbols in a signature—functions from symbols to “real” mathematical objects. In the presence of a change in signature  $\sigma : \Sigma \rightarrow \Sigma'$ , a  $\Sigma'$ -algebra can interpret symbols in  $\Sigma$  by first applying  $\sigma$  and interpreting the resulting  $\Sigma'$ -symbol; hence we “precompose”  $A'$  by  $\sigma$  to obtain a  $\Sigma$ -algebra. Note that the direction is reversed; we are taking  $\Sigma'$ -algebras to  $\Sigma$ -algebras using  $\sigma : \Sigma \rightarrow \Sigma'$ . Now is a good time to note the contravariance of the model functor in the definition of an institution: if  $\sigma : \Sigma \rightarrow \Sigma'$  then  $\text{Mod}(\sigma) : \text{Mod}(\Sigma') \rightarrow \text{Mod}(\Sigma)$ .

The following pair of definitions explain how we may build more complex expressions, which we will call *terms*, out of the basic symbols of a signature.

**Definition 7.** A set of variables for a signature  $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$  is an  $S$ -indexed set.

**Definition 8.** A term over a signature  $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$  with variables in  $X$  is defined inductively as follows.

- A variable  $x \in X_s$  is a term of sort  $s$ .
- A constant symbol  $C \in \mathcal{F}_{\text{nil},s}$  is a term of sort  $s$ .
- If  $|w| = n > 0$ , then given terms  $t_1 : w_1, \dots, t_n : w_n$  and a function symbol  $F \in \mathcal{F}_{w,s}$ , the expression  $F(t_1, \dots, t_n)$  is a term of sort  $s$ .

Terms explain how sorted variables and symbols may be put together. For example, let  $x : \text{elem}$  and  $s : \text{stack}$  be two variables; then  $\text{push}(x, s) : \text{stack}$  is a valid term; as is  $\text{push}(x, \text{push}(x, s))$ . But, for example,  $\text{pop}(x)$  is not since  $x$  has the wrong sort.

With terms and algebras defined, all that is left is to define first-order sentences.

**Definition 9.** Let  $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$  be a signature. The sentences of first-order logic are built from the logical symbols  $=, \rightarrow, \neg, \wedge, \vee, \forall, \exists$ . The atomic sentences are

- $u = v$  for terms  $u$  and  $v$  with the same sort; and
- $P(t_1, \dots, t_n)$  for any predicate symbol  $P \in \mathcal{P}_w$  and terms  $t_i$ .

The sentences in general are defined inductively as follows:

- Any atomic sentence  $\phi$  is a sentence.
- The expressions  $\neg\phi, \phi \rightarrow \psi, \phi \wedge \psi, \phi \vee \psi, \forall x. \phi$  and  $\exists x. \phi$ , for any sentences  $\phi, \psi$  and variable  $x$ , are all sentences.

We can now write sentences like  $\forall x. \forall s. \text{pop}(\text{push}(x, s)) = s$ . The interpretation of first-order sentences is defined by induction on the sentence structure. We will give a more precise account in Sect. 4.

### 3 Institutions in Coq

Coq is an interactive proof assistant for higher-order logic based on a dependent type theory called the *calculus of inductive constructions* (CIC). Dependent type theories allow for extremely elegant representation of complex mathematical objects such as those found in category theory, institution theory, universal algebra, etc. All of the work presented here is formalised fully in Coq.

We depend on a formalization of category theory by John Wiegley [20]. Morphisms between objects  $a$  and  $b$  are denoted  $a \rightsquigarrow b$ , and functors between categories  $C$  and  $D$  are denoted  $C \dashrightarrow D$ . The generic form of an institution can be defined directly as a dependent record.

```

Class Institution :=
{ Sig : Category ;
  Sen : Sig --> SetCat ;
  Mod : Sig^op --> SetCat ;
  interp : ∀ (Σ : Sig), Mod Σ -> Sen Σ -> Prop ;
  sat : ∀ (Σ Σ' : Sig) (σ : Σ ~> Σ') (φ : Sen Σ) (M' : Mod Σ'),
    interp M' (fmap[Sen] σ φ) <-> interp (fmap[Mod] σ M') φ }.

```

Here `interp` refers to the semantic entailment relation  $\models$  for an institution and `SetCat` refers to the category of sets, which here just means the category of Coq types and functions. The term `fmap` is short for “functor map” and describes the action of a functor on morphisms. For the purposes of this paper, we implement so-called *set/set institutions* [11], in which the target of `Mod` is `Set`, the category of sets, and not `Cat`, the category of all small categories.

Our focus for this paper is on an instantiation of this object to *FOPEQ*, the institution for first-order predicate logic, and *EVT*, the institution for Event-B defined in [6]. Since *EVT* builds on *FOPEQ*, we will begin with *FOPEQ* and work up.

## 4 First-Order Logic in Coq

We partially build upon a formalization of multi-sorted universal algebra in Agda [9], though we deviate in many of the details. As we define objects in Coq, we will make reference back to their mathematical definitions from Sect. 2.1. We do not show everything, only what we deem crucial to follow the basic idea of the formalization.

### 4.1 Representing FOL

Signatures (cf. Definition 3) are represented by a dependent record, mirroring the mathematical definition exactly.

```

Record Signature :=
{ Sorts : Type ;
  Funcs : list Sorts -> Sorts -> Type ;
  Preds : list Sorts -> Type }.

```

An algebra (cf. Definition 5) for a signature needs to interpret sorts as Coq types and the function and predicate symbols as Coq functions with the right type.

For this we use *heterogeneous lists*—henceforth *h-lists*—following Gunther *et al.* [9]. A heterogeneous list can contain elements of different types, as distinguished from a homogeneous list which contains only elements of a single type. Our definition of h-lists comes from Chlipala’s CPDT [4], where the reader can find a more detailed description of the implementation details.

Let  $\mathcal{U}$  be a universe of types. Given an index type  $I : \mathcal{U}$ , a list  $w : \text{List}(I)$  and a type family  $A : I \rightarrow \mathcal{U}$  which selects for each  $i : I$  a type  $A_i : \mathcal{U}$ , we can build

a h-list  $v : \text{HList}(A, w)$  which contains  $|w|$  elements and where the  $i$ th element of  $v$ , denoted  $v_i$ , has the type  $A_{w_i}$ . For example, if  $w = [\mathbb{N}, \text{bool}, \text{string}]$  and if  $A$  is the identity, then  $\langle 3, \text{true}, \text{'hello'} \rangle$  would be a valid h-list of type  $\text{HList}(A, w)$ . Another example, more pertinent to our discussion: Consider again our running example `stackSig`. Let  $I = \{\text{elem}, \text{stack}\}$ , let  $w = [\text{elem}, \text{stack}]$ , and let  $A : I \rightarrow \mathcal{U}$  be defined by  $\text{elem} \mapsto \mathbb{N}$  and  $\text{stack} \mapsto \text{List}(\mathbb{N})$ . Then  $\langle 2, [3, 4] \rangle$  would be a valid term of type  $\text{HList}(A, w)$ .

A h-list is a concrete implementation of a kind of dependent  $n$ -tuple; that is to say,  $\text{HList}(A, w)$  is a concrete Coq encoding of the dependent sum  $\sum_i A(w_i)$ . Now, let  $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$  be a signature, let  $F \in \mathcal{F}_{w,s}$  and let  $A$  be a  $\Sigma$ -algebra. Since  $\text{HList}(A, w) \rightarrow A(s) \cong \sum_i A(w_i) \rightarrow A(s)$ , we should interpret  $A(F)$  as a function  $\text{HList}(A, w) \rightarrow A(s)$ .

```
Record Algebra  $\Sigma$  :=
{ interp_sorts : Sorts  $\Sigma$  -> Type ;
  interp_funcs w s : Funcs  $\Sigma$  w s -> HList interp_sorts w -> interp_sorts s ;
  interp_preds w : Preds  $\Sigma$  w -> HList interp_sorts w -> Prop }.
```

We are so far no different from Gunther *et al.* [9]. Our first deviation is in the definition of variables and terms (cf. Definition 8).

```
Inductive Term : Sorts  $\Sigma$  -> Type :=
| var s : member s  $\Gamma$  -> Term s
| term w s : Funcs  $\Sigma$  w s -> HList Term w -> Term s.
```

Variables (cf. Definition 7) are not represented here by members of an indexed set; instead they are dependent de Bruijn indices—see CPDT chapter nine [4]. The `member` type is exactly as it appears there; a term  $i : \text{member}(s, \Gamma)$  can be thought of as a constructive proof that  $s$  appears at index  $i$  in the list  $\Gamma$ . By defining variables this way, we can quite easily define quantifiers which correctly track the locations of free variables, as we will see.

Signature morphisms (cf. Definition 4) are the cornerstone of institution theory; much of the implementation depends on this definition.

```
Record SigMorphism  $\Sigma$   $\Sigma'$  :=
{ on_sorts : Sorts  $\Sigma$  -> Sorts  $\Sigma'$  ;
  on_funcs w s : Funcs  $\Sigma$  w s -> Funcs  $\Sigma'$  (map on_sorts w) (on_sorts s) ;
  on_preds w : Preds  $\Sigma$  w -> Preds  $\Sigma'$  (map on_sorts w) }.
```

No surprises here, but note that we must translate the sorts in `on_funcs` and `on_preds` using `on_sorts`. With this we can now define `reduct` algebras (cf. Definition 6).

```
Definition ReductAlgebra ( $\sigma$  : SigMorphism  $\Sigma$   $\Sigma'$ ) (A' : Algebra  $\Sigma'$ ) : Algebra  $\Sigma$  :=
{| interp_sorts := interp_sorts A'  $\circ$   $\sigma$  ;
  interp_funcs :=  $\lambda$  w s F, interp_funcs A' (on_funcs  $\sigma$  F)  $\circ$  reindex  $\sigma$  ;
  interp_preds :=  $\lambda$  w P, interp_preds A' (on_preds  $\sigma$  P)  $\circ$  reindex  $\sigma$  |}.
```

Note that the function `reindex` is computationally the identity but converts between the equivalent types  $\mathbf{HList}(A \circ f, w)$  and  $\mathbf{HList}(A, \text{map } f w)$ .

We can now start to build the syntactic and semantic structure of first-order sentences. The syntax is as follows.

```

Inductive FOL : list (Sorts  $\Sigma$ ) -> Type :=
| Forall  $\Gamma$  s : FOL (s ::  $\Gamma$ ) -> FOL  $\Gamma$ 
| Exists  $\Gamma$  s : FOL (s ::  $\Gamma$ ) -> FOL  $\Gamma$ 
| Equal  $\Gamma$  s : Term  $\Sigma$   $\Gamma$  s -> Term  $\Sigma$   $\Gamma$  s -> FOL  $\Gamma$ 
| Pred  $\Gamma$  w : Preds  $\Sigma$  w -> HList (Term  $\Sigma$   $\Gamma$ ) w -> FOL  $\Gamma$ 
| ...

```

We omit the other connectives since their definitions are straightforward. Syntactically, a quantifier accepts as argument a sentence in which at least one variable appears free and binds it. If  $\psi$  is a sentence with context  $s :: \Gamma$ , then the sentence  $Q_s. \psi$  is a sentence with context  $\Gamma$ , where  $Q$  is either quantifier. Formally, we have the following syntactic formation rule:

$$\frac{s :: \Gamma \vdash \phi}{\Gamma \vdash Q_s. \phi}$$

To interpret a first-order sentence, we must decide what the logical symbols mean and what values the free variables will get. If  $\theta$  is an environment providing values for the variables in  $\Gamma$ , then we denote the semantic interpretation of a sentence  $\phi$  with free variables from  $\Gamma$  by an algebra  $A$  with environment  $\theta$  by  $A \models^\theta \phi$ . Precisely, in the case of the quantifiers, we have

$$\begin{aligned} A \models^\theta \text{Forall}_s(\psi) &\quad \text{iff} \quad \text{for all } x \in A_s \text{ we have } A \models^{x, \theta} \psi \\ A \models^\theta \text{Exists}_s(\psi) &\quad \text{iff} \quad \text{there exists } x \in A_s \text{ such that } A \models^{x, \theta} \psi \end{aligned}$$

This setup makes the definition of the semantic entailment relation relatively painless. (The triple-colon operator denotes the cons function for h-lists.)

```

Fixpoint interp_fol (A : Algebra  $\Sigma$ ) ( $\varphi$  : FOL  $\Sigma$   $\Gamma$ ) ( $\theta$  : HList A  $\Gamma$ ) : Prop :=
match  $\varphi$  with
| Forall s  $\psi$  =>  $\forall$  x : A s, interp_fol A  $\psi$  (x ::  $\theta$ )
| Exists s  $\psi$  =>  $\exists$  x : A s, interp_fol A  $\psi$  (x ::  $\theta$ )
| Equal u v => eval_term A u  $\theta$  = eval_term A v  $\theta$ 
| Pred P ts => interp_preds A P (map_eval_term A ts  $\theta$ )
| ...

```

The institution *FOPEQ* requires closed first-order sentences, i.e. sentences of the form  $\psi : \mathbf{FOL}(\Sigma, \text{nil})$ ; hence  $A \models \psi$  will really mean  $\text{interp\_fol}(A, \psi, \text{hnil})$ .

The relation above relies on two mutually-defined term evaluation functions, for which we use the `coq-equations` library [18].



```

Equations eval_term (A : Algebra  $\Sigma$ ) (t : Term  $\Sigma \Gamma s$ )
  : HList A  $\Gamma \rightarrow$  A s := {
  eval_term _ (var i)  $\theta$  := HList.nth  $\theta$  i ;
  eval_term A (term F args)  $\theta$  := interp_funcs A F (map_eval_term A args  $\theta$ )
} where map_eval_term (A : Algebra  $\Sigma$ ) (args : HList (Term  $\Sigma \Gamma$ ) w)
  : HList A  $\Gamma \rightarrow$  HList A w := {
  map_eval_term _ () _ := () ;
  map_eval_term A (t :: ts)  $\theta$  :=
  eval_term A t  $\theta$  :: map_eval_term A ts  $\theta$  }.

```

On variables, it looks up the right value in the environment. On terms, it interprets the function symbol with the given algebra and calls itself on the function symbol's arguments. It is possible to write this function (and others) without coq-equations, but this gives the best computational behaviour and plays relatively nicely with the proofs.

## 4.2 Proofs and Proof Strategy

There are some more definitions that are crucial for proving the satisfaction condition for first-order predicate logic. The following mutually-defined functions promote signature morphisms to term translations:

```

Equations on_terms ( $\sigma$  : SigMorphism  $\Sigma \Sigma'$ ) (t : Term  $\Sigma \Gamma s$ )
  : Term  $\Sigma'$  (map  $\sigma \Gamma$ ) ( $\sigma s$ ) := {
  on_terms  $\sigma$  (var i) := var (reindex_member  $\sigma$  i) ;
  on_terms  $\sigma$  (term F args) := term (on_funcs  $\sigma$  F) (map_on_terms  $\sigma$  args)
} where map_on_terms ( $\sigma$  : SigMorphism  $\Sigma \Sigma'$ ) (args : HList (Term  $\Sigma \Gamma$ ) w)
  : HList (Term  $\Sigma'$  (map  $\sigma \Gamma$ )) (map  $\sigma w$ ) := {
  map_on_terms  $\sigma$  () := () ;
  map_on_terms  $\sigma$  (t :: ts) := on_terms  $\sigma$  t :: map_on_terms  $\sigma$  ts }.

```

Applying a signature translation to a variable amounts only to a reindexing; all the work is happening at the type level, but the underlying “number”  $i$  doesn't change. To apply a signature translation to a term, we just apply it to the function symbol and then apply it to all its arguments. Promoting this a level higher to first-order sentences is a simple matter, since the sentence structure will be ignored by signature morphisms.

We will also need to define a custom induction principle for terms; the induction principle automatically generated by Coq is too weak because it is missing a hypothesis in the case where the term has the form  $F(t_1, \dots, t_n)$ ; namely that the predicate  $P : \prod_i A_i \rightarrow \text{Prop}$  holds for all  $t_1, \dots, t_n$ . In Coq this is represented by  $\text{HForall } P \langle t_1, \dots, t_n \rangle$ .

```
Context (Σ : Signature) (Γ : list (Sorts Σ)).
Context (P : ∀ s, Term Σ Γ s -> Prop).
```

```
Hypothesis var_case : ∀ s (m : member s Γ), P s (var m).
Hypothesis term_case :
  ∀ w s (F : Funcs Σ w s) (args : HList (Term Σ Γ) w),
  HForall P args -> P s (term F args).
```

```
Equations term_ind' s (t : Term Σ Γ s) : P s t := {
  term_ind' s (var i) := var_case _ i ;
  term_ind' s (term F args) := term_case _ _ F args (map_term_ind' _ args)
} where map_term_ind' w (args : HList (Term Σ Γ) w) : HForall P args := {
  map_term_ind' s {} := I ;
  map_term_ind' s (t ::: ts) := conj (term_ind' _ t) (map_term_ind' _ ts) }.
```

*Proofs Involving Indexed Types.* Consider how to define the composition of two signature morphisms  $\sigma$  and  $\tau$ . Doing so directly will result in a type-level mismatch between  $\text{map } \tau (\text{map } \sigma w)$  and  $\text{map } (\tau \circ \sigma) w$ . Of course, these terms are propositionally equal via the proof  $p = \text{map\_map } \sigma \tau w$ ; so we need to mention this to Coq at the point of definition. As an example, here is the definition of the composition of two first-order signature morphisms, simplified for readability.

```
Definition comp_FOSig
  (τ : SigMorphism B C) (σ : SigMorphism A B) : SigMorphism A C :=
{| on_sorts := τ ∘ σ ;
  on_funcs w s F :=
    rew (map_map σ τ w) in (on_funcs τ (on_funcs σ F)) ;
  on_preds w P :=
    rew (map_map σ τ w) in (on_preds τ (on_preds σ P)) |}.
```

Many definitions in our developments take a similar form. Proofs of most propositions involving such terms should follow by computation and induction on the involved identity proofs—an *identity proof* being a proof of the form  $p : x = y$ . We call upon a range of tactics and rewriting strategies for identity proofs, many of which are defined in Coq.Init.Logic and some of which come from the homotopy type theory [19] Coq developments, specifically Basics/PathGroupoids.v.

Proofs about terms caused the most consternation. Using the following lemma,

```
Lemma map_on_terms_hmap (σ : SigMorphism Σ Σ') (ts : HList (Term Σ Γ) w) :
  map_on_terms σ ts = reindex σ (HList.map (on_terms σ) ts).
```

we can write `map_on_terms` in terms of `hmap` and `on_terms`; this exposes `reindex`, which we may convert into `rew` using some combination of the following two lemmas.

```
Lemma reindex_id (v : HList A is) :
  reindex idmap v = rew (map_id is)^ in v.
```

```
Lemma reindex_comp (f : I -> J) (g : J -> K) (v : HList (A ◦ (g ◦ f)) is) :
  rew (map_map f g is) in reindex g (reindex f v) = reindex (g ◦ f) v.
```

This process pulls out the hidden identity proof such that it may be combined with others. We then required the lemma

```
Lemma map_ext_HForall (f g : ∀ i, A i -> B i) (v : HList A w) :
  HForall (λ i x, f i x = g i x) v <-> hmap f v = hmap g v.
```

for converting the hypothesis generated by our custom induction principle for terms into a useful rewrite rule.

There is one more trick we employ, and for which we must assume proof irrelevance. Often the subject of an identity proof is of the form  $x :: xs$ , as in, for example,  $p' : \text{map id } (x :: xs) = x :: xs$ . If one has a proof  $p : \text{map id } xs = xs$ , then in fact  $\text{f\_equal } (\text{cons } x) p$  is also a proof that  $\text{map id } (x :: xs) = x :: xs$ . By proof irrelevance,  $p'$  and  $\text{f\_equal } (\text{cons } x) p$  are themselves equal, but the point is that the latter form has useful structure that we can exploit. This is not always necessary—often the `simplify_eqs` tactic is enough—but we found it indispensable in proofs which required more careful rewriting of identity proofs.

*The Proof of Satisfaction.* Throughout the process, we identified at least one non-obvious lemma required for the proof of satisfaction for first-order logic.

**Lemma 1.** *Let  $\sigma : \Sigma_1 \rightarrow \Sigma_2$  be a signature morphism, let  $t_1$  be a  $\Sigma_1$ -term with  $\Sigma_1$ -context  $\Gamma_1$ , and let  $A_2$  be a  $\Sigma_2$ -algebra. Let  $\theta : \text{HList}(A_2|_\sigma, \Gamma_1)$  be a valuation of the variables in  $\Gamma_1$ . Then*

$$A_2^{\sigma(\theta)}(\sigma(t_1)) = (A_2|_\sigma)^\theta(t_1)$$

Since  $\theta$  is a h-list, the action of  $\sigma$  on  $\theta$  is just a reindexing; hence we obtain  $\sigma(\theta) : \text{HList}(A_2, \text{map } \sigma \Gamma_1)$ . The specific requirement generated by the proof of satisfaction, for one of the atomic sentences,  $t_1 = t_2$ , is

$$A' \models^{\sigma(\theta)} (\sigma(t_1 = t_2)) \quad \text{iff} \quad (A'|_\sigma) \models^\theta (t_1 = t_2)$$

Lemma 1 is a strict strengthening of this requirement, since it shows in fact that the terms under analysis are equal. This lemma handles the atomic sentences; the other cases follow without much trouble.

## 5 Formalizing *EVT*

Readers should consult the backmatter of [16] for a summary of the Event-B language by Thai Son Hoang. Not much, if any, familiarity with the system will be required beyond what we describe here. Event-B machines consist, at the

most basic level, of discrete state-transitions called *events*. Our approach is to use first-order sentences to represent updates to the machine state; for example, the variable-update statement  $x := x + 1$  is written as a first-order sentence  $x' = x + 1$ . The unprimed variables represent the state of the machine before an event, the primed variables represent the state of the machine after an event.

We will formally describe a more generic institution for Event-B than is presented in Farrell [6], dropping event names from the representation. We will call this institution *EVT* where there is no room for confusion. We found that defining *EVT* without event names results in simplified constructions and gives us more room for defining potential extensions to *EVT*—but, crucially, without changing the details of the proof of satisfaction. For a short account of a formalization that matches Farrell’s more closely, see [13].

First, we’ll define signatures and signature morphisms for *EVT*.

**Definition 10.** An *EVT*-signature  $\hat{\Sigma}$  is a 3-tuple  $\langle \Sigma, X, X' \rangle$  where  $\Sigma$  is a first-order signature and  $X$  and  $X'$  are  $\text{Sorts}(\Sigma)$ -indexed sets, such that  $(-)' : X \rightarrow X'$  is an equivalence.

**Definition 11.** An *EVT*-signature morphism  $\hat{\sigma} : \hat{\Sigma}_1 \rightarrow \hat{\Sigma}_2$  consists of a first-order signature morphism  $\sigma : \Sigma_1 \rightarrow \Sigma_2$  and two variable morphisms  $\text{on\_vars} : X_1 \rightarrow X_2$  and  $\text{on\_vars}' : X'_1 \rightarrow X'_2$  such that the following diagram commutes.

$$\begin{array}{ccc} X_1 & \xrightarrow{\text{on\_vars}} & X_2 \\ (-)'\uparrow & & \uparrow(-)' \\ X'_1 & \xrightarrow{\text{on\_vars}'} & X'_2 \end{array}$$

In all cases where not otherwise specified, the *EVT*-signature  $\hat{\Sigma}$  is given by  $\langle \Sigma, X, X' \rangle$ .

A standard construction in institution theory is the *signature extension*. We add variables by adding them directly into the signature as constant function symbols.

**Definition 12.** Let  $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$  be a first-order signature and let  $X$  be an  $S$ -indexed set. The expansion of  $\Sigma$  by  $X$  is a first-order signature  $\Sigma + X$  which is equal to  $\Sigma$  everywhere except on the constant function symbols;  $\Sigma + X$  has constant symbols  $\mathcal{F}_{\text{nil},s} + X_s$ , for each  $s \in S$ .

To model signatures of the form  $\Sigma + X$ , we need only expand a given  $\Sigma$ -algebra by a valuation  $X \rightarrow A$ .

**Definition 13.** Let  $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$  be a first-order signature and let  $A$  be a  $\Sigma$ -algebra. Let  $X$  be an  $S$ -indexed set of variables and let  $\theta : X \rightarrow A$  be a valuation of variables. The expansion of  $A$  by  $\theta$  is a  $(\Sigma + X)$ -algebra  $A^\theta$ , which behaves like  $A$  on symbols from  $\Sigma$  and takes variables  $x \in X_s$  to  $\theta(x) \in A_s$ .

**Definition 14.** A  $\hat{\Sigma}$ -model  $M$  is a 3-tuple  $\langle A, \theta, \theta' \rangle$ , where  $A$  is a  $\Sigma$ -algebra and  $\theta : X \rightarrow A$  and  $\theta' : X' \rightarrow A$  are valuations of variables.

To illustrate what we have so far, consider the following example. A model for a  $(\Sigma + X + X')$ -sentence consists of a  $\Sigma$ -algebra  $A$  and two valuations of the variables  $\theta : X \rightarrow A$  and  $\theta' : X' \rightarrow A$ , usually referred to as a  $\Sigma$ -states— $\theta$  is the pre-state and  $\theta'$  is the post-state. One possible model for the sentence  $x' = x + 1$  consists of the usual algebra for natural numbers, a pre-state  $x \mapsto 2$ , and a post-state  $x' \mapsto 3$ . One possible model for the sentence  $s' = \text{push}(x, s)$  consists of an algebra for a stack of characters, a pre-state  $x \mapsto e$ ,  $s \mapsto [v, t]$ , and a post-state  $s' \mapsto [e, v, t]$ . Here,  $x'$  can consistently be assigned anything. If we wish to avoid this, we can assume that sentences  $\psi$  which don't mention a given primed variable  $x'$  are really shorthand for  $\psi \wedge (x' = x)$ .

Let us formally define the sentences for *EVT*. Note that  $\text{FOSen}(\Sigma)$  denotes the set of all first-order  $\Sigma$ -sentences.

**Definition 15.** *Let  $\hat{\Sigma}$  be an EVT-signature. A  $\hat{\Sigma}$ -sentence is either an initialization sentence  $\text{Init}(\phi)$  where  $\phi \in \text{FOSen}(\Sigma + X')$ , or an event sentence  $\text{Event}(\phi)$  where  $\phi \in \text{FOSen}(\Sigma + X + X')$ .*

Initialization sentences constrain the range of possible initial states for a machine; often only one such state is possible. There is no previous state yet, so any initialization sentence is built over  $\Sigma + X'$ . Event sentences explain how an event updates the state, and therefore can access both pre- and post-variables; thus event sentences are built over  $\Sigma + X + X'$ .

Finally, let's define the semantic entailment relation for *EVT*.

**Definition 16.** *Let  $\hat{\Sigma}$  be an EVT-signature,  $M = \langle A, \theta, \theta' \rangle$  a  $\hat{\Sigma}$ -model, and  $\psi$  an EVT-sentence. We define  $M \models \psi$  by induction on  $\psi$ :  $M \models \text{Init}(\phi)$  if  $A^{\theta'} \models \phi$ ; and  $M \models \text{Event}(\phi)$  if  $A^{\theta+\theta'} \models \phi$ .*

## 5.1 Representing EVT

We have a much easier job here than we did for first-order predicate logic since *EVT* builds directly on *FOPEQ*. We rely on a couple of major first-order constructions. First, we will define signature extensions by a set of variables (cf. Definition 12). Note that  $\text{Indexed I}$  is the category of indexed types  $I \rightarrow \mathcal{U}$ .

```

Definition SigExpand ( $\Sigma : \text{FOSig}$ ) ( $X : \text{Indexed (Sorts } \Sigma)$ ) :  $\text{FOSig} :=
\{ | \text{Sorts} := \text{Sorts } \Sigma ;
  \text{Funcs} := \lambda w s, \text{match } w \text{ with}
    | [] => \text{Funcs } \Sigma [] s + X s
    | _ => \text{Funcs } \Sigma w s
  \text{end} ;
  \text{Preds} := \text{Preds } \Sigma | \}.$ 
```

The main part of an algebra expansion (cf. Definition 13) is given by the following function; no other part of the algebra is changed.

```

Equations alg_exp_funcs
  (A : Algebra  $\Sigma$ ) ( $\theta : X \rightsquigarrow A$ ) w s (F : Funcs (SigExpand  $\Sigma$  X) w s)
  : HList A w -> A s :=
  alg_exp_funcs [] s (inr C) :=  $\lambda \_, \theta$  s C ;
  alg_exp_funcs [] _ (inl F) := interp_funcs A F ;
  alg_exp_funcs (_ :: _) _ F := interp_funcs A F .

```

*EVT*-signatures (cf. Definition 10) are represented exactly as they are given mathematically.

```

Record EvtSignature :=
{ base :=> Signature ;
  Vars  : Sorts base -> Type ;
  Vars' : Sorts base -> Type ;
  prime_rel : Primed Vars Vars' }.

```

Here `prime_rel` is the proof that  $(-)' : X \rightarrow X'$  is an equivalence.

For *EVT* signature morphisms (cf. Definition 11), we simply define `on_vars'` in terms of `on_vars` to simplify matters.

```

Record EvtSigMorphism  $\Sigma$   $\Sigma'$  : Type :=
{ on_base :=> SignatureMorphism  $\Sigma$   $\Sigma'$  ;
  on_vars  : Vars  $\Sigma \rightsquigarrow$  Vars  $\Sigma' \circ$  on_base }.

```

```

Definition on_vars' ( $\sigma : \text{EvtSigMorphism } \Sigma \Sigma'$ ) : Vars'  $\Sigma \rightsquigarrow$  Vars'  $\Sigma' \circ \sigma :=
  \lambda s x, \text{prime } (\sigma s) (\text{on\_vars } \sigma (\text{unprime } s x))$ .

```

*EVT*-models (cf. Definition 14) and *EVT*-sentences (cf. Definition 15) also offer no surprises.

```

Record EvtModel  $\Sigma$  :=
{ base_alg :=> Algebra  $\Sigma$  ;
  env  : Vars  $\Sigma \rightsquigarrow$  base_alg ;
  env' : Vars'  $\Sigma \rightsquigarrow$  base_alg }.

```

```

Inductive EVT  $\Sigma$  : Type :=
| Init  : Sen[FOL] (SigExpand  $\Sigma$  (Vars'  $\Sigma$ )) -> EVT  $\Sigma$ 
| Event : Sen[FOL] (SigExpand  $\Sigma$  (Vars  $\Sigma \otimes$  Vars'  $\Sigma$ )) -> EVT  $\Sigma$ .

```

Finally, the semantic entailment relation for *EVT* (cf. Definition 16) defers directly to entailment for *FOPEQ*.

```

Definition interp_evt (M : EvtModel  $\Sigma$ ) ( $\phi : \text{EVT } \Sigma$ ) : Prop :=
  match  $\phi$  with
  | Init  $\psi$  => AlgExpansion (base_alg M) (env' M)  $\models \psi$ 
  | Event  $\psi$  => AlgExpansion (base_alg M) (join_vmmaps (env M) (env' M))  $\models \psi$ 
  end.

```

Here, `join_vmmaps` stitches two valuations  $\theta : X \rightarrow M$  and  $\theta' : X' \rightarrow M$  (with the same target) into  $\Theta : X + X' \rightarrow M$ .

## 5.2 Proofs and Proof Strategy

Most of the tricks we needed for first-order logic apply just as well here. The main additional proof strategy emerged while proving equality for dependent records.

As an example, let's consider *EVT* signature morphisms. To prove that two *EVT* signature morphisms are equal, we need to prove that they are equal componentwise—the first-order signature morphisms have to agree everywhere, as do the two variable morphisms. The proofs that the variable morphisms are equal *appear* at first to depend on the proof that the base first-order signature morphisms are equal. But actually, that's not strictly the case; we only need to know that the first-order signature morphisms agree on sorts to prove that the two variable morphisms are equal.

We can write custom equality lemmas which state the dependencies between proofs more precisely. Here is one such lemma for *EVT* signature morphisms.

```
Lemma eq_evt_sig_morphism (σ σ' : EvtSigMorphism Σ Σ')
  (p' : on_sorts σ = on_sorts σ')
  (p : on_base σ = on_base σ')
  (q : rew [λ σs, Vars Σ ~> Vars Σ' ∘ σs] p' in
    on_vars σ = on_vars σ')
  : σ = σ'.
```

Here we build a proof that two signature morphisms are equal from proofs that they are equal at each of their components. Note that  $q$  depends on  $p'$  only, and not  $p$ . Normally the dependency is on  $p$ —but  $p'$  is typically much simpler than  $p$  and is all that is necessary. Often  $p'$  is `refl`, meaning `rew` computes away, simplifying the proofs considerably.

Most other constructions and proofs revolved around signature and model extensions. The following was the main non-trivial lemma which we identified while proving the satisfaction condition for *EVT*. Note that the following holds for *any* indexed sets  $X_1$  and  $X_2$  and *any* function  $f : X_1 \rightarrow X_2 \circ \sigma$ .

**Lemma 2.** *Let  $\sigma : \Sigma_1 \rightarrow \Sigma_2$  be a first-order signature morphism, let  $f : X_1 \rightarrow X_2 \circ \sigma$  be a variable morphism, let  $A_2$  be a  $\Sigma_2$ -algebra, and let  $\theta_2 : X_2 \rightarrow A_2$  be a valuation of variables. Then*

$$(A_2|_{\sigma})^{\theta_2 \circ f} = (A_2^{\theta_2})|_{\sigma + f}$$

We're taking some liberties with the notation. Note that  $\theta_2 \circ f$  is a shorthand for  $\lambda s, x. \theta_2(\sigma(s), f(s, x))$  and  $\sigma + f : \Sigma_1 + X_1 \rightarrow \Sigma_2 + X_2$  is a shorthand for the extension of  $\sigma$  by  $f$ .

The proof of satisfaction itself proceeds by two cases, both of which are essentially the same, and both of which rely on the satisfaction condition for *FOPEQ* and Lemma 2, with  $f$  instantiated to different maps in each.

## 6 Conclusion

We have detailed the most important points of our formalisation of two institutions in Coq: the institution *FOPEQ* for first-order predicate logic, and the

institution *EVT* for Event-B. According to the `cloc` tool<sup>1</sup> we have over 3,000 significant lines of Coq developments, not including the many experimental or variant implementation attempts. Initial progress was slow, but the overall approach was successful; the satisfaction condition is fully formalized for both—with some difficulty for *FOPEQ*, but with far greater ease for *EVT*. Furthermore, both institutions have many reusable components which will aid in the construction of other concrete institutions and with proving their satisfaction conditions. Proofs involving indexed types in Coq are notoriously difficult, but we suspect for the purposes of our formalism that they are all difficult in the same way, so that the lessons we learn here can be applied more generally.

Having a formal framework for defining institutions continues to be useful in our own work. Indeed, we have already begun applying it to the problem of integrating linear-time temporal logic with Event-B specifications. We have also defined some institution-independent constructions not covered here, specifically modal and linear-time temporal logics over an arbitrary institution.

We intend in the future to add more concrete institutions to this framework; to show that both *FOPEQ* and *EVT* have the amalgamation property; to build more institution-independent constructions; to improve proof automation for institutions; and to define and verify some institution (co)morphisms. This work could also, in time, become a fully formal basis for the work already done for the HETS tool for heterogeneous specification.

## References

1. Amato, G., Maggesi, M., Parton, M., Brogi, C.P.: Universal Algebra in UniMath (2020). <https://arxiv.org/abs/2007.04840>
2. Burstall, R.M., Goguen, J.A.: The semantics of clear, a specification language. In: Bjørner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 292–332. Springer, Heidelberg (1980). [https://doi.org/10.1007/3-540-10007-5\\_41](https://doi.org/10.1007/3-540-10007-5_41)
3. Capretta, V.: Universal algebra in type theory. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 131–148. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-48256-3\\_10](https://doi.org/10.1007/3-540-48256-3_10)
4. Chlipala, A.: Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013). <http://adam.chlipala.net/cpdt/>
5. Coq Development Team: The Coq Proof Assistant. <https://coq.inria.fr/>
6. Farrell, M.: Event-B in the Institutional Framework: Defining a Semantics, Modularisation Constructs and Interoperability for a Specification Language. Ph.D. thesis, National University of Ireland Maynooth (2017). <http://mural.maynoothuniversity.ie/9911/>
7. Goguen, J.A., Burstall, R.M.: Introducing institutions. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 221–256. Springer, Heidelberg (1984). [https://doi.org/10.1007/3-540-12896-4\\_366](https://doi.org/10.1007/3-540-12896-4_366)
8. Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. *J. ACM* **39**(1), 95–146 (1992). <https://doi.org/10.1145/147508.147524>

<sup>1</sup> <https://github.com/AlDanial/cloc>.



9. Gunther, E., Gadea, A., Pagano, M.: Formalization of universal algebra in Agda. *Electron. Notes Theor. Comput. Sci.* **338**, 147–166 (2018). <https://doi.org/10.1016/j.entcs.2018.10.010>
10. Knapp, A., Mossakowski, T., Roggenbach, M., Glauer, M.: An institution for simple UML state machines. In: Egyed, A., Schaefer, I. (eds.) *FASE 2015*. LNCS, vol. 9033, pp. 3–18. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46675-9\\_1](https://doi.org/10.1007/978-3-662-46675-9_1)
11. Mossakowski, T., Goguen, J., Diaconescu, R., Tarlecki, A.: What is a logic? In: *Logica Universalis*, pp. 111–133. Birkhäuser Basel (2007)
12. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, HETS. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-71209-1\\_40](https://doi.org/10.1007/978-3-540-71209-1_40)
13. Reynolds, C.: Formalizing the institution for Event-B in the coq proof assistant. In: Raschke, A., Méry, D. (eds.) *ABZ 2021*. LNCS, vol. 12709, pp. 162–166. Springer, Cham (2021). [https://doi.org/10.1007/978-3-030-77543-8\\_17](https://doi.org/10.1007/978-3-030-77543-8_17)
14. Riehl, E.: *Category Theory in Context*. Dover Modern Math Originals, Dover Publications, Aurora (2017)
15. Roggenbach, M.: CSP-CASL—a new integration of process algebra and algebraic specification. *Theor. Comput. Sci.* **354**(1), 42–71 (2006). <https://doi.org/10.1016/j.tcs.2005.11.007>
16. Romanovsky, A., Thomas, M. (eds.): *Industrial Deployment of System Engineering Methods*. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-33170-1>
17. Sannella, D., Tarlecki, A.: *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Series, Springer, Heidelberg (2012). <https://doi.org/10.1007/978-3-642-17336-3>
18. Sozeau, M.: Equations: a dependent pattern-matching compiler. In: Kaufmann, M., Paulson, L.C. (eds.) *ITP 2010*. LNCS, vol. 6172, pp. 419–434. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14052-5\\_29](https://doi.org/10.1007/978-3-642-14052-5_29)
19. *Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics* (2013). <https://homotopytypetheory.org/book>. Institute for Advanced Study
20. Wiegley, J.: *Category Theory in Coq*. <https://github.com/jwiegley/category-theory>