MACHINE-ASSISTED PROOFS *for* INSTITUTIONS

Conor Reynolds

Submitted for the degree of
Doctor of Philosophy

Maynooth University
Department of Computer Science

OCTOBER 2023

Head of Department Dr Joseph Timoney
Supervised by Prof Rosemary Monahan

Contents

All the genuine, deep delight of life is in showing people the mud-pies you have made; and life is at its best when we confidingly recommend our mud-pies to each other's sympathetic consideration.

— J. M. Thorburn

Abstract

Institution theory is the abstract study of logical systems using category theory. The theory has expanded since the '80s to encompass and relate a wide range of concrete logical systems. But inducting particular logical systems into the theory can be tiresome—often straightforward, but repetitive, technical, and prone to error. Worse, such work suffers from the very problem which motivates abstraction: we find ourselves proving the same things over and over again.

In this thesis we will describe a framework for easing the process of constructing institutions and verifying their properties. The goal is primarily to expedite the verification of those proof obligations most commonly involved in constructing institutions. We start by encoding the institution for first-order predicate logic, turning later to the semantics for the Event-B modelling method. We then explore a particular logic combination of Event-B and linear temporal logic, as well as some interesting institution-independent constructions that enable generic logic combinations and translations—all of which will serve as useful work in its own right, but also as a demonstration of the framework.

Acknowledgements

My PhD experience was typical by 2019 standards but is nowadays thankfully atypical. Most of it was spent alone in my bedroom waiting for the COVID-19 pandemic to end—presuming it ever would. I'm happy to report that both the pandemic and my PhD did end. And much like the pandemic, it only ended because it was decided that enough had been done and we should all get on with our lives.

I couldn't have done this without the support of my parents, Richard and Carol, who were unfortunately locked in the house with me for most of my PhD but nevertheless avoided going insane. My supervisor, Rosemary Monahan, endured an unacceptable number of rambling emails and DMs with admirable patience. Without her I would not have had the mental strength to complete the work. Marie Farrell too was an especially helpful presence in the latter stages of the PhD, and who also endured much of my ceaseless palaver. My friends have been with me through everything and have helped me more than they know. I thank you all.

Thanks as well to Markus Roggenbach and Tom Dowling, my PhD examiners. Their comments have greatly informed the thesis you see now.

I hereby abandon this work so that I may release it. If you spot any typos, don't tell me.

Note on the Type

The body and headings are set respectively in Equity[1] and Concourse,[2] both designed by Matthew Butterick. The mathematics is set in a combination of fonts. It is primarily a modified copy of Donald Knuth's Computer Modern by Chuanren Wu.[3] The blackboard bold font is Antonis Tsolomitis's New Computer Modern,[4] the script font is STIX Two,[5] and the sans serif math font is a copy of Latin Modern that I modified to be a little thicker in FontForge. The monospace font is Source Code Pro.[6] Does it all somehow work together? Judge for yourself.

I agree with C. Wu's assessment of Computer Modern. It's hard to find a superior free OTF math font with all its features. But it is undeniably emaciated and does not in my view work well as a body font. The reasons for the gaunt appearance are likely due to more accurate modern printing—printed CM, as in Knuth's *Digital Typography*, did not suffer from this problem.[7] But even thickened it is exhausting to read for long stretches and is at any rate overused. Why not try something new?

ONE

Introduction and Motivation

In this chapter, we'll motivate the research and explain our approach. We'll outline the rest of the thesis, which properly begins in CHAPTER 2.

1.1 *Research Summary*

Institutions capture the abstract notion of 'logical system'. Introduced by Joseph Goguen and Rod Burstall, they have a reasonably long history by now, making their first official appearance in [GB84] and fleshed out in [GB92]. Their mathematical history goes back even further to Goguen and Burstall's semantics for Clear [BG79] and K. Jon Barwise's axioms for abstract model theory [Jon74], in which a proto-satisfaction condition appears as the 'translation axiom'. Institution theory allows us to study logical systems in the general case, and by showing that particular logical systems are institutions we may import all of the benefits of the general theory.

A 'logical system' provides a syntax for writing down certain sorts of sentences, as well as a way to interpret the meaning of those sentences. For example: temporal logic eases reasoning about time; modal logic eases reasoning about counterfactuals; linear logic eases reasoning about resources; Hoare logic eases reasoning about imperative programming languages; the list goes on. It is common in the literature to find minor variations on these logics, so that we do not just have one 'Temporal Logic', but really a *family* of logics that all reason about temporal proper-

ties in slightly different ways. This has the effect of ballooning the total number of distinct logical systems out there in the literature.

A key observation of Goguen and Burstall in [GB92] was that certain results common in applications are actually independent of the particular logic. If so, and if institutions are the right abstraction of 'logical system', then we expect these general results to be true of institutions with only minimal extra assumptions.

Another advantage is that institutions put logic translation front and center, allowing us to formally relate multiple similar logics, to impute a semantics to one logic via translation into another more established logic, or even to state semantics-preserving translations from one logic into another, permitting the reuse of existing theorem provers to check the veracity of translated sentences.

Much of the motivation for this work is in Marie Farrell's thesis [Far17], in which she defines an institution—and therefore a semantics—for Event-B. An important goal, successfully achieved, was to prove that her institution for Event-B, called EVT, satisfied the *amalgamation property*. This showed that EVT supports important modularisation constructs.

But constructing institutions by hand and proving their satisfaction conditions, much less using them in practice, can be anything from tedious to onerous—a task reserved for those few experts who know institution theory intimately, preventing wider adoption. To address this problem, we formalise the theory of institutions in the Coq proof assistant [Coq23]. The central research question is this: How can we best harness Coq to ease the process of constructing institutions and verifying their properties?

This thesis contributes a formalisation of the theory of institutions in Coq, which also serves as a framework for the future development of institutions in an interactive proof assistant. First, we prove the satisfaction condition for a key institution: first-order predicate logic with equality, $FOPEQ$, the first such machine-assisted proof of which we are aware. Much of the machinery we develop to prove this turns out

to be of great general use, in no small part because many institutions are either similar to or directly build upon *FOPEQ*. This means we can make relatively short work of *EVT* and it's logical core *ACT* which we define in CHAPTER 4. Better still, because *EVT* and *ACT* are based on a rather generic notion of 'variable update', the proofs involved for their satisfaction conditions turn out to be of general use for any institution in which there appears a designated set of state variables—this includes *LTL* and *MacEVT*, two institutions that we will define in CHAPTER 5. The framework was not altogether trivial to develop, but with the foundational work done, we have hopes that it can be easily and fruitfully iterated upon.

We tested a range of potential new institutions in this framework, the majority of which do not appear in this thesis because they were faulty for one reason or another—and we were able to identify those faults quickly within the framework. The speed of development of new institutions has increased dramatically. And though the framework still needs work to become more generally useful, especially to those without a lot of experience with Coq, I believe that it is only a matter of time.

1.1.1 *Publications*

There are a number of peer-reviewed publications related to this work. The first is a 4 page ABZ 2021 conference paper [Rey21] which gives a very brief overview of our formalisation of Farrell's institution for Event-B in Coq. The next is a 16 page TASE 2022 conference paper [RM22] which gives much more details about the formalisation of first-order logic in Coq, containing elements of the second, third, fourth, and final chapters of this thesis. We expanded on this in a 2024 SCP journal paper [RM24]. The work in CHAPTER 5 is new and has not yet been published anywhere.

1.2 *Related Work*

Though the goals of the thesis broadly have been heretofore unexplored in the literature, there is a wealth of related ideas.

Related formalisations We build directly on the work done by Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano formalising multi-sorted universal algebra in Agda [GGP18]. We also note some other work in this direction in Coq by Venanzio Capretta [Cap99], by Gianluca Amato, Marco Maggesi, Maurizio Parton and Cosimo Perini Brogi [Ama+20], as well as Andreas Lynge and Bas Spitters [LS19]. Our approach most resembles Gunther's but deviates to enable a particular implementation of quantifiers.

Alternatives to institutions One alternative to institutions for the semantics of imperative programming languages is Unifying Theories of Programming (UTP) [HJ98], based on set theory and first-order logic. This sounds somewhat specific for the stated goals of the thesis, but programs and specifications do not really represent a difference in kind as much as a difference in degree—specifications are typically loose constraints on an execution trace, and programs are typically tight constraints on an execution trace. Event-B, the tool that forms a key example in CHAPTERS 4 & 5, is based on a similar conception of programs and specifications; a central feature of Event-B is its support for syntactical machine refinement from 'abstract' machines ('specifications') to 'concrete' machines ('programs') [Abr10]. While this could be a sensible approach to a semantics for Event-B, we are doing more than just formalising a semantics for Event-B.

Alternatives to Coq The Coq proof assistant [Coq23] is a popular choice for formalising programming language semantics—and as indicated, programming languages are not so different from logics as far as institutions are concerned. It is based on a dependent type theory called the calculus of inductive constructions [CH88], which allows for elegant representations of complicated mathematical objects.

We considered Isabelle/HOL as an alternative to Coq; the book *Concrete Semantics* [NK14] is a textbook on Isabelle where the second part focuses on formalising the syntax and semantics of programming languages. There are multiple formalisations of category theory in Isabelle/HOL, for example [OKe04]. There is even Isabelle/UTP [FZW15], a mechanisation of UTP in Isabelle/HOL. We ultimately went with Coq due to its dependent type system and the more convincing and complete formalisations of category theory—but there is of course no reason why Isabelle/HOL would not work.

Other proof assistants worth mentioning are Agda [Agd], which has better support for dependently typed programming but no tactic language; Lean [dMou+15], which is based on a very similar type theory to Coq but which is much newer and less mature than Coq; and the experimental Arend proof assistant [Are] for homotopy type theory.

Logical frameworks such as Dedukti[8] or Twelf,[9] both based on the $\lambda\Pi$-calculus [CD07], can be used to express theories and logics—but it seems to me that we need to be able to express the logics themselves as objects for manipulation (specifically institutions), and it is not clear that these frameworks are any advantage in that regard over Coq. Interaction between our developments and Dedukti, however, would certainly be a topic for future consideration.

There is also Florian Rabe's scalable module system for mathematical theories, MMT, a 'mathematical knowledge management' system [RK13]. But this is more of a system for expressing and representing logical frameworks in a uniform setting than a tool that can help us to show that the representations themselves are sound.

Alternatives to Wiegley's category theory library Institutions are based on category theory, so it is important that we build on existing work formalising category theory in Coq. We use John Wiegley's category theory library [Wie14], but there are many others—most notably the category theory developments in the Coq-HoTT library[10] based on homotopy type theory [UFP14]. These developments are excellent, but the biggest advantage of Wiegley's library is that it is *axiom free*, meaning we're free

to add whatever axioms we think we need as long as they are consistent with the Coq kernel. The homotopy type theory developments assume the univalence axiom, which contradicts axioms we use here, namely *uniqueness of identity proofs*. More on that in SECTION 2.2.3.

1.3 *Validity of Machine-Assisted Proofs*

Proof assistants are software, and like all software prone to error. Bugs in Coq are caught and fixed all the time. Some surely lurk there still. How then are we to trust that the proof checker is correct? And even if we do trust it, how can we be sure that the encoding of mathematics in the proof assistant accurately models mathematical reality? We must not believe the fantasy that we can encode a mathematical statement in a proof assistant, see that the proof checker declares it correct, and assume that the corresponding natural language fact is true.

Proof assistants are just that: assistants. There is no way to expunge humans from the verification and validation process. Trust in a proof assistant is not blind but collaborative—we work *with* the assistant to achieve a peculiar approximation of certainty. But since we nevertheless regard certification by a proof assistant as somehow increasing our certainty, we must be deferring a degree of real work to it. We are trusting to some extent that it upholds its end of the deal. Does it?

If Coq can be trusted for any reason, especially compared to other interactive proof assistants, it is because it has been well-tested.* It is one of the most popular proof assistants out there—and with so many experts relying on it for its correctness, bugs in the system are caught and fixed quickly. There is even the MetaCoq project which formalises Coq itself within Coq [Soz+20].

Well and good, but bugs remain, or so we must assume. How does this avoid vitiating work done in Coq? We must ask whether our proofs

*This is discussed in [Rog+22]. It might seem besides the point—we can either trust it (because it's correct) or we can't (because it's not)—but trust is a function of what we *know* and not what *is*, and more eyeballs means more opportunities to spot problems. There can be no absolute certainty here since we are comparing an idea with its representation. We should take all the evidence we can get.

leverage those bugs in a meaningful way. This is a bigger problem for automated proof assistants than interactive ones because automated systems may typically choose freely among the available reasoning principles. Suppose I add a contradiction to my list of axioms:

```
Axiom oops : ∀ (A : Prop), A ∧ ¬A.
```
1.3.1

Many automated provers will quickly discover that they can dramatically simplify their proofs by employing this remarkable axiom! But in an interactive setting, there's no reason to believe that we might by mere virtue of this axiom's existence accidentally prove falsehoods. We perfectly well understand that this axiom introduces an inconsistency, and we're in the driver's seat, so we avoid it. Moreover, when writing proofs we often know that something is wrong when a conclusion seems to not 'really follow' from the assumptions. Humans require not just proofs, but *reasons*, and proofs that leverage contradictions are often bereft of reasons.

This example is artificial, but consider this example of a real bug in Agda. In Agda 2.5.3, irrelevant projections made it possible to prove $\mathsf{true} \equiv \mathsf{false}$ [Gil+19].[†] But the derivation of this inconsistency is somewhat delicate, so it is hard to see how many existing developments in Agda could possibly leverage it or inconsistencies akin to it in their own proofs unless they make very heavy and complex use of irrelevant projections. And even so, does that imply that their work is unsalvagable, relying *crucially* upon such an inconsistency? It's possible, but unlikely.

The more serious problem in my view is that of encoding. We do not deal directly with mathematical reality in a proof assistant, we deal with its representation in a formal system—in Coq's case, a dependent type theory. Consider sets. In type theory, sets are not primitives, so we must represent them using type-theoretic machinery. Usually, but not always, a set $S \subseteq U$ is represented by its membership predicate, since predicates have a natural representation as type families $S : U \to \mathsf{Prop}$, where $x \in S$ in set theory is written $S(x)$ in type theory. But is this

[†]See also this GitHub issue: github.com/agda/agda/issues/543.

really the same object? If we prove something for such a type family, have we proved it for its 'equivalent' set? If not, what can we say we've proved? Does this representation import all of the baggage of sets? Can we distinguish sets and classes? Do we need to? More generally, if a statement involving sets is true in Coq, is its 'equivalent' statement in set theory true too? These issues of encoding are quite serious, and we will deal with them as they arise in the thesis. (See [Bar10] for a set-theoretical model of the calculus of constructions, which goes some way to investigating the questions posed above, and see SECTION 2.2 for a more detailed discussion of the relationship between set theory and type theory.)

For now let's say this: Our encoding will try to be more relaxed in general—we assume very little about the components that make up the theory. We only assume what we need to prove the results we are interested in, while still retaining the basic shape of the theory. This helps to make the formalisation believable. If I assume much fewer things than necessary to flesh out the full mathematical object, then it is much easier for me to believe that what is true for the representation is true for the mathematical object.

There's one more modelling decision worth discussing. Any embedding of an object logic in a proof assistant (the meta-logic) can be shallow or deep. A shallow embedding represents the object logic's sentences directly as sentences in the meta-logic. A deep embedding represents the object logic's sentences as data in the meta-logic. Shallow embeddings are simpler to implement than deep embeddings—we get semantic interpretation of sentences 'for free' since we are representing object sentences directly in the model—but make it very difficult to prove metatheorems about the logic. We are quite directly interested in proving metatheorems about the object logics, so we choose a deep embedding.

1.4 *How to Read This Thesis*

This thesis is written in such a way that it should be possible to jump around as you wish. Every chapter except CHAPTER 6 builds directly on the previous chapter. Of course, if you are so inclined, I would recommend reading it start to finish, but I'm not so naive as to believe that anyone besides my thesis examiners and a select few dedicated proofreaders will do that. (I appreciate your sacrifice.) Your time is valuable and I encourage you to pick around.

There is an index on page 139 as well as special indices for mathematical symbols and Coq tactics on pages 133 and 135 respectively.

Almost everything in the thesis has been labelled, including every single code listing—though not every listing is explicitly referenced in the text. Since there are many—*many*—listings in this thesis, I had to invent a less obtrusive method of captioning:

```
Fixpoint factorial (n : nat) : nat :=                    1.4.1
  match n with
  | 0 => 1
  | S k => S k * factorial k
  end.
```

Notice the small number on the top right? This will be referenced as LISTING 1.4.1, with a numbering system independent of definitions, theorems, etc. If you're reading this in a PDF reader, any text set in small-caps is a clickable cross-reference.

There are occasional references to the GitHub repository where all the work is stored:

> https://github.com/ConorReynolds/coq-institutions/tree/thesis

Those references look like this: Core/Basics.v#L24. If you're reading this in a PDF reader, this text is hyperlinked to the repository. If not, interpret this as the instructions 'navigate to line 24 in the file Basics.v in the directory theories/Core'. Make sure you navigate to the 'thesis' tag, otherwise the code there may not reflect the code in the thesis.

Everything else has been done in a conventional manner and doesn't require explanation. I'll give a brief synopsis of the thesis now so you can skip ahead to whatever takes your fancy.

CHAPTER 2 Mathematical background—category theory, institution theory, universal algebra, dependent type theory. If you're comfortable with all of these topics then I certainly encourage you to skip (or at least skim) most of this chapter and return to it as needed.

CHAPTER 3 Covers the basics of the formalisation and encodes the institution for first-order logic in Coq called *FOPEQ*, including a proof of its satisfaction condition. Develops most of the proof techniques required for the remainder of the thesis.

CHAPTER 4 Builds upon the previous chapter to construct a simplified institution for Event-B called *ACT*, including a proof of its satisfaction condition. Introduces a few more proof techniques for working with institutions with state.

CHAPTER 5 Iterates on the ideas so far introduced and constructs an institution combining a bespoke linear temporal logic with Event-B using a duplex construction and a novel institution for Event-B called *MacEVT*, which directly encodes the machine semantics of Event-B.

CHAPTER 6 Discusses some interesting institution-independent constructions, formalises some institution-independent model theory, the amalgamation property, deductive systems for institutions, etc. This chapter is part concrete results and part speculation about future directions, and shows (we hope) the full scope of what is possible within the framework.

CHAPTER 7 Concludes the thesis and discusses more speculative future research directions.

TWO

Mathematical Background

The required mathematical background for this thesis is not so extensive. For completeness, we record it here and refer back to it later.

We will cover basic category theory in SECTION 2.1. Then we discuss the relevant differences between set theory and type theory in SECTION 2.2. Institutions will be defined in SECTION 2.3, followed by universal algebra and first-order logic in SECTION 2.4. Feel free to skip this chapter and return as needed.

2.1 *Category Theory*

The category-theoretic prerequisites for most of this thesis are relatively light, requiring only basic knowledge of categories, functors, and natural transformations. I recommend Emily Riehl's freely available *Category Theory in Context* [Rie17] or Steve Awodey's *Category Theory* [Awo10] for a more detailed account.

If you are not familiar with category theory, it might help to think of categories, functors, and natural transformations in the terms outlined in Joseph Goguen's 'Categorical Manifesto' [Gog91]. Therein Goguen describes various 'categorical dogmas'—informal heuristics which can help to identify appropriate applications of categorical concepts, and which also double up as useful mental models of those concepts. We will paraphrase the relevant dogmas as appropriate.

2.1.1 *Definition* [Rie17]. A *category* $\mathscr{C}$ consists of a collection of objects and a collection of morphisms, such that:

- Each morphism f has a *domain* object a and *codomain* object b, written $f : a \to b$. We will write $\mathrm{dom}(f) = a$ and $\mathrm{cod}(f) = b$.
- For each object a there is an identity morphism $1_a : a \to a$.
- For any morphisms $f : a \to b$ and $g : b \to c$, with $\mathrm{cod}(f) = \mathrm{dom}(g)$, there is a composite morphism $g \circ f : a \to c$.

These data are subject to the following laws:

- *Identity*: For any $f : a \to b$, we have $1_b \circ f = f \circ 1_a = f$.
- *Composition*: The composites $f \circ (g \circ h)$ and $(f \circ g) \circ h$ are equal.

Each kind of mathematical object generally corresponds to a category consisting of those objects and structure-preserving morphisms between them—sets and set-functions, groups and group homomorphisms, signatures and signature morphisms (as we will soon see), and so on.

The expression $x \in \mathscr{C}$ means that x is an object of $\mathscr{C}$. When the category is clear, we will write $f : a \to b$ for a morphism in $\mathscr{C}$ from a to b; when it is not clear, we will write $f \in \hom_{\mathscr{C}}(a, b)$. We will not use the diagrammatical order $f \fatsemi g = g \circ f$ of function composition since the presentation is not thereby improved.

Two standard categories appear throughout the paper—the category Set of all sets and set-functions, and the category Cat of all (small) categories and functors between them.

2.1.2 *Definition* [Rie17]. A *functor* $F : \mathscr{C} \to \mathscr{D}$ between categories $\mathscr{C}$ and $\mathscr{D}$ consists of

- an object $Fx \in \mathscr{D}$ for each object $x \in \mathscr{C}$; and
- a morphism $F(f) : Fa \to Fb$ for each morphism $f : a \to b$,

subject to the following laws:

- *Identity*: $F(1_a) = 1_{Fa}$.
- *Composition*: $F(g \circ f) = F(g) \circ F(f)$.

Functors are mappings between categories that preserve the categorical structure. It may be best in this context to think of a functor from the category of x's into the category of y's as a construction of y's over x's [Gog91]. For example, later on we will define a 'sentence functor' from a category of signatures to Set, which will give us two things: the set of sentences built over any given signature, and a way to lift signature morphisms to sentence morphisms that preserve the sentence structure.

2.1.3 *Definition* [Rie17]. A *natural transformation* $\eta : F \Rightarrow G$ between functors $F, G : \mathscr{C} \to \mathscr{D}$ consists of a collection of morphisms $\eta_a : Fa \to Ga$ in $\mathscr{D}$ for each $a \in \mathscr{C}$, such that, for any $f : a \to b$ in $\mathscr{C}$, the following diagram commutes.

$$\begin{array}{ccc} Fa & \xrightarrow{\eta_a} & Ga \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ Fb & \xrightarrow[\eta_b]{} & Gb \end{array}$$

The $\eta_a : Fa \to Ga$ are called the *components* of η.

Natural transformations represent a relationship or mapping between two functors (constructions). We will see natural transformations much later in CHAPTERS 5 & 6 when we discuss institution morphisms. The simplest context to see them in action is with duplex institutions, discussed in SECTION 5.6. More complex categorical constructions will be defined as we need them.

2.2 *Set Theory and Type Theory*

The implied (or explicit) foundation for much practical mathematics is set theory. Institution theory is no exception. But since we intend to encode institutions in a dependent type theory, in which the notion of 'set' is not basic, we should explain how to translate some set-theoretic constructions to type theory. Mike Shulman's blog post on this topic is a good introductory read [Shu13]. Further information can be found in the homotopy type theory book [UFP14].

Set theory and its accompanying logic can be thought of as a formal deductive system with a single judgement: 'A is true', or 'A has a proof' [UFP14]. Note that the judgement 'A is true' is distinguished from the proposition A—the first is a statement in the metalanguage outside the deductive system, but the second is a statement in the deductive system itself. Type theory, however, has two judgements: $a : T$, meaning 'a has the type T' and $a \triangleq b : T$, meaning 'a and b are definitionally equal terms of type T'. We will discuss them in turn.

2.2.1 *Set membership and typing judgements*

What does $x \in S$ mean informally? That depends. Suppose I write

> Let $n \in \mathbb{N}$.

This is a declaration that n is a natural number. It is not a proposition that can be true or false. But now suppose $E \subseteq \mathbb{N}$ is the set of all even numbers, and I write

> If $n \in E$ then $n + 1 \notin E$.

Here '$n \in E$' is a proposition. It can be true or false.

ZFC set theory does not distinguish these usages—all membership is propositional. For instance, consider the following statement:

$$\forall n \in \mathbb{N}.\, P(n)$$

ZFC set theory regards membership here as a predicate, so that this really means 'for all things n, if n is a natural number, then $P(n)$'. But type theory and structural set theory would regard this sentence as asserting that P is a property of all natural numbers, and which is perhaps senseless to apply to objects that are not natural numbers. So in type theory, the declaration $n \in \mathbb{N}$ is translated as the judgement $n : \mathbb{N}$, which means that n *has the type* $\mathbb{N}$. The propositional form $n \in E$ regards the subset $E \subseteq \mathbb{N}$ as a predicate $E : \mathbb{N} \to \mathsf{Prop}$ and evaluates it at n—that is, $n \in E$ is equivalent to $E(n)$.

The blurring of these uses can complicate formalisation efforts for subfields of mathematics that make heavy use of set-theoretic language, such as institution theory and model theory. We will rarely use the set membership symbol '$\in$' and the typing judgement symbol ':' interchangeably, but confusion should not result anyway—most of us are used to the implicit dual usage of the set membership symbol.

2.2.2 *Propositional and judgemental equality*

In the same way that type theory separates the set-theoretic notion of membership into two distinct concepts, it has two notions of equality split along similar lines—one definitional, one propositional. *Definitional* or *judgemental equality* is, unsurprisingly, true 'by definition', but it's better thought of as metatheoretical equality living outside the formal system in question. This kind of equality is written $x \triangleq y$. It does not make sense to ask if $x \triangleq y$ is true in the *internal* language of type theory, but it (or its negation) may be assumed in the metalanguage.

Propositional equality, however, is type theory's internal notion of equality. It is represented by a type called the *identity type*, written

$$\mathsf{Id}_A : A \to A \to \mathcal{U}$$

where $\mathcal{U}$ is a universe of types. If a term p inhabits the type $\mathsf{Id}_A(x, y)$, it means that x and y are propositionally equal, and p is usually thought of as a 'proof' that x and y are equal, or as 'evidence' of their equality. Instead of $\mathsf{Id}_A(x, y)$ we will write $x =_A y$, or simply $x = y$ if the types are clear from context. To prove that two objects are equal using type theory, we must form the judgement $p : x = y$ by constructing the proof term p using the internal rules of type theory. (We'll see how this works in SECTION 3.1.2.)

The identity type is defined like any other inductive type. The sole introduction rule is

$$\mathsf{refl} : \prod_{a:A} (a = a)$$

This rule allows us to construct the reflexive proof $\mathsf{refl}_a : a = a$ for any $a : A$.* The induction principle for the identity type encodes the notion of *indiscernibility of identicals*. It says that for every type family $P : A \to \mathcal{U}$, there is a function

$$\mathsf{eq\_rect}_P : \prod_{(x,y:A)} (x = y) \to P(x) \to P(y)$$

which is defined by induction to be

$$\mathsf{eq\_rect}_P(x, x, \mathsf{refl}_x, t) \triangleq t \tag{2.1}$$

The shorthand for $\mathsf{eq\_rect}_P(x, y, p, t)$ in Coq is rew $[P]$ p in t, or simply rew p in t if the type family P can be inferred from context. We will use this notation from now on. This is the mechanism by which propositional equalities can be used to perform rewrites. This principle of the indiscernibility of identicals, encoded in eq_rect, is what enables us to substitute equals for equals in type theory.

This information will be important to keep in mind for later. To see the immediate relevance of this information, go to SECTION 3.1.2. The examples there will be useful for understanding the preceding discussion—but before you go, keep in mind the following common notation for identity proofs: If $p : x = y$ then we'll write $p^{-1} : y = x$ for the reverse proof. If $q : y = z$ is another proof, then we'll write $p \cdot q : x = z$, encoding transitivity. The Coq notations are respectively eq_sym p and eq_trans p q, which will occasionally appear in Coq proofs (like for example LISTINGS 3.3.7 & 4.3.5).

2.2.3 *Axioms in type theory*

This is a practical formalisation effort, so we will adopt many axioms common in type theory with no scruples. None of the following are

*A common presumption is that the members of a type are exactly those obtained by repeatedly applying constructors, which if true would imply that refl is the only possible identity proof. While true for simpler types like $\mathbb{N}$, this is false for the identity type Id. See [UFP14, §5.8] for a discussion on this point. It is better in general to think of a type's constructors as freely generating its elements.

possible to prove in an intensional type theory like Coq's CIC, but all are consistent with the Coq kernel and each other. The main assumptions are—

- *Dependent function extensionality*—proving $f = g$ for functions $f, g : \prod_{x:A} B(x)$ amounts to proving, for all $x : A$, that $f(x) = g(x)$.
- *Invariance by substitution of reflexive equality proofs*—given $t : A_x$ and a proof $p : x = x$, we have $t = \mathsf{rew}\ p\ \mathsf{in}\ t$. This is equivalent to the axiom of *uniqueness of reflexive identity proofs*, which says that any $p : x = x$ is itself propositionally equal to refl_x. We will refer to this axiom as UIP.
- *Proof irrelevance*—for any proposition $p : \mathsf{Prop}$ and proofs $a, b : p$, there is a proof $a = b$; that is, all proofs of a proposition are equal. This is stronger than the preceding axiom and used sparingly, but it's occasionally more practical.

The latter two axioms contradict the *univalence axiom* of homotopy type theory [Gil+19]. It has rarely been practical to avoid their use, but we have tried anyway in places to test compatibility with homotopy type theory should that become a relevant concern for us.

For those unfamiliar with type theory the last axiom listed above may seem a kind of nonsense. What can it possibly mean for two proofs of a proposition to be equal? How can we consistently say that any two such proofs are equal?

In type theories, a proposition P is just a certain kind of type, and a proof is a term of that type. If $t : P$ is a term of type P then *a fortiori* P is true, since that is precisely what it means for a proposition to be true in type theory—it is inhabited by a term; it has a proof. Since proofs of propositions are terms like any other, they can appear in data structures. It is not uncommon to construct dependent pairs (x, p) where x is some value and p is a proof involving x. Let's provide an explicit example.

Consider the problem of defining non-empty lists in type theory. One way to represent them is as the type of all pairs (ℓ, p), where ℓ is a regular list which may be empty, but p is a proof that ℓ is in fact not empty. How should we decide if two non-empty lists are equal? Usually, to prove $(a, b) = (c, d)$, we need to prove both $a = c$ and $b = d$. But in the case of the non-empty list, it is sufficient to prove $a = c$ because the second component, the proof, is just there to certify that the lists are not empty. It's irrelevant to the question. But we can't simply ignore the proof component. So the next best alternative is to *declare* that all proofs of the same proposition are equal—that is the principle of proof irrelevance. That way $(\ell, p) = (\ell', p')$ requires us to prove $\ell = \ell'$ alone, with $p = p'$ true by fiat.

2.3 *Institution Theory*

Institutions are based on category theory. Return to SECTION 2.1 for the necessary prerequisites if you are not familiar.

2.3.1 *Definition* [GB84]. An *institution* consists of

- a category Sig of signatures;
- a sentence functor $\mathsf{Sen} : \mathsf{Sig} \to \mathsf{Set}$;
- a model functor $\mathsf{Mod} : \mathsf{Sig}^{\mathsf{op}} \to \mathsf{Cat}$; and
- a semantic entailment relation $\models_\Sigma \subseteq |\mathsf{Mod}(\Sigma)| \times \mathsf{Sen}(\Sigma)$ for each $\Sigma \in \mathsf{Sig}$,

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

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

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

A functor $F : \mathscr{C} \to \mathscr{D}$ can allow us to imagine that morphisms in $\mathscr{C}$ are also valid morphisms of $\mathscr{D}$ which preserve some structure. This

permits the unambiguous use of a signature morphism $\sigma : \Sigma \to \Sigma'$ to act on sentences and models without mentioning the ambient functor. Hence we will write $\sigma(\varphi)$ for $\mathsf{Sen}(\sigma)(\varphi)$, and $M|_\sigma$ for $\mathsf{Mod}(\sigma)(M)$.

The components of an institution can be split into two broad types: signatures and sentences encode syntactical information; models and the semantic entailment relation encode semantic information. We can also split them along 'logical' lines: signatures and models are the non-logical data; sentences and the semantic entailment relation are the logical data. By 'logical' we mean that the data involves truth or falsity. The number 'three' and the sentence 'Pass the salt' are not logical insofar as they cannot be true or false or do not involve deciding truth or falsity. Something like '$2 + 2 = 5$' or 'My dog is asleep', however, *are* logical in this sense and can be true or false. (Wilfred Hodges calls these 'declarative sentences' [Hod01, §2].) Thus we have our succinct description of the four components—

	non-logical	*logical*
syntax	Sig	Sen
semantics	Mod	$\models$

But what about the satisfaction condition?

The category of signatures does not just contain signatures; it also contains *signature morphisms*. These morphisms, on their face, represent a simple change in notation. But in practice they are key enablers of a wide range of more complex logic translations, such as institution morphisms and comorphisms. Such translations can make precise the intuitive notion that propositional logic is contained in predicate logic, itself contained in first-order logic, and even to enable *logic combinations*.

It also gives rise to the *satisfaction condition*, which is an account of how the four previously introduced components ought to interact under translation. Consider a sentence $\varphi \in \mathsf{Sen}(\Sigma)$ and a model $M' \in \mathsf{Mod}(\Sigma')$. In the presence of a signature translation $\sigma : \Sigma \to \Sigma'$, there are two possibilities—either apply the signature translation to the sentence φ and ask if $M' \models \sigma(\varphi)$, or apply the signature translation to the model

M' and ask if $M'|_\sigma \models \varphi$. The satisfaction condition says that it doesn't matter—they are logically equivalent. The following picture might help.

$$\begin{array}{ccccc} \Sigma' & & M' & \models_{\Sigma'} & \sigma(\varphi) \\ {\scriptstyle\sigma}\uparrow & & \downarrow & & \uparrow \\ \Sigma & & M'|_\sigma & \models_\Sigma & \varphi \end{array}$$

The situation is sometimes given by the aphorism

Truth is invariant under change of notation [GB92]

but this can be a little misleading—really we want to say that the meaning of the *sentences* should be preserved by signature translation. Moreover, the satisfaction condition asserts that the meaning of a sentence does not depend on the context in which it appears [ST11]. This is actually false for some logical systems, but is true for the logical systems considered in this thesis.

2.4 *First-Order Predicate Logic*

In this section we'll provide a formal account of multi-sorted universal algebra and first-order predicate logic in preparation for a formal encoding in Coq in CHAPTER 3. See Sannella and Tarlecki's *Foundations of Algebraic Specification* [ST11, Chapter 1 and §4.1] for an extended account, and see [Hod93, Chapter 1] for another presentation of similar material in a single-sorted setting. We will only focus on what we need for the formalisation.

First, some preliminary mathematical definitions.

2.4.1 *Definition* [ST11]. An *S-indexed set* X is a family of sets indexed by S. This is formally a function $X : S \to \mathsf{Set}$ which defines for each $s \in S$ a set $X(s)$, which we denote X_s.

2.4.2 *Definition.* The set of all finite sequences of elements from a set A is denoted $\mathsf{List}(A)$. This set comes equipped with the usual constructors

$\mathsf{nil} : \mathsf{List}(A)$ and $\mathsf{cons} : A \to \mathsf{List}(A) \to \mathsf{List}(A)$. The ith element of a list ℓ, where $0 \leq i < \mathsf{len}(\ell)$, is written ℓ_i.

Now we're ready to define first-order signatures.

2.4.3 *Definition* [ST11]. A *first-order signature* is a 3-tuple $\langle S, \mathscr{F}, \mathscr{P} \rangle$ where S is a set of sorts, $\mathscr{F}$ is a $(\mathsf{List}(S) \times S)$-indexed set of function symbols, and $\mathscr{P}$ is a $\mathsf{List}(S)$-indexed set of predicate symbols.

A sort is a label attached to data which describes what kind of data it is. In modern parlance, a sort is a type, but we will not call them types to avoid confusion with types in type theory.

A function symbol $F \in \mathscr{F}_{w,s}$ is said to have arity w and result sort s; a predicate symbol $P \in \mathscr{P}_w$ is said to have arity w and no result sort. Arities in single-sorted contexts are natural numbers which denote the number of arguments a function or predicate symbol has, but in a multi-sorted context we additionally need to specify the sorts of the arguments.

Since the signature is usually clear from context, we will write $F : \prod_i w_i \to s$ for function symbols, and $P : \prod_i w_i \to \mathsf{Prop}$ for predicate symbols. If a function symbol C has arity nil and result sort s, then it is called a *constant symbol* and we denote it $C : s$.

Here Prop has no semantic import, it's merely a syntactic marker for a predicate symbol to distinguish it from function symbols. In [ST11], predicate symbols are written $P : \prod_i w_i$, but this clashes a little with the notation for constant symbols and seems to suggest that P is a tuple.

Let's make this concrete with an example. Let $\mathsf{stack\_sig}$ be a simple signature consisting of the handful of symbols required to describe a stack data structure. It has two sorts elem and stack; three function symbols

$$\begin{aligned} \mathsf{empty} &: \mathsf{stack} \\ \mathsf{push} &: \mathsf{elem} \times \mathsf{stack} \to \mathsf{stack} \\ \mathsf{pop} &: \mathsf{stack} \to \mathsf{stack} \end{aligned}$$

and a single predicate symbol

$$\mathsf{isEmpty} : \mathsf{stack} \to \mathsf{Prop}$$

Note that although these symbols and their sorts *suggest* a meaning, none has been given. They can't be 'used' because they are symbols and have no meaning. We are free to choose what the symbols mean as long as the meanings are congruent. Each such choice is called an *algebra* or *first-order model*.

2.4.4 *Definition* [ST11]. An *algebra* A for a signature $\Sigma = \langle S, \mathscr{F}, \mathscr{P} \rangle$ consists of three functions $\langle A_{sorts}, A_{funcs}, A_{preds} \rangle$, 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 $F \in \mathscr{F}_{w,s}$, we have $A(F) : A_{w_1} \times \cdots \times A_{w_n} \to A_s$; and
- for any $P \in \mathscr{P}_w$, we have $A(P) \subseteq A_{w_1} \times \cdots \times A_{w_n}$.

Note that, following [ST11], we allow empty carrier sets.

2.4.5 *Definition* [ST11]. An *algebra homomorphism* $h : A \to B$ between two Σ-algebras A and B is a function $h : \prod_s A_s \to B_s$ satisfying, for any $F \in \mathscr{F}_{w,s}$ (with $|w| = n$) and values $a_1 : A_{s_1}, \ldots, a_n : A_{s_n}$,

$$h(A(F)(a_1, \ldots, a_n)) = B(F)(h(a_1), \ldots, h(a_n))$$

Consider again our running example stack_sig. We could interpret the sort elem as the set $\mathbb{N}$ of natural numbers, and the sort stack as the set $\mathsf{List}(\mathbb{N})$ of lists of natural numbers. We could then interpret the function symbols empty, push, and pop as the usual functions on lists:

$$\begin{aligned} \mathsf{nil} &: \mathsf{List}(\mathbb{N}) \\ \mathsf{cons} &: \mathbb{N} \times \mathsf{List}(\mathbb{N}) \to \mathsf{List}(\mathbb{N}) \\ \mathsf{tail} &: \mathsf{List}(\mathbb{N}) \to \mathsf{List}(\mathbb{N}) \end{aligned}$$

The predicate symbol isEmpty can be interpreted as the predicate

$$\{ s \mid s = \mathsf{nil} \} \subseteq \mathsf{List}(\mathbb{N})$$

The next step is to explain how to form more complex terms out of the symbols of a signature.

2.4.6 *Definition* [ST11]. A set of *variables* for a signature $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$ is an S-indexed set. Given some ambient pool of variables X, we will write $x : s$ to mean $x \in X_s$, mirroring the notation for constant symbols.

2.4.7 *Definition* [ST11]. 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 $C \in \mathcal{F}_{\mathsf{nil},s}$ is a term of sort s.
- Let w be a list of sorts such that $|w| = n$ and $n > 0$. Given terms $t_1 : w_1, \ldots, t_n : w_n$ and a function symbol $F \in \mathcal{F}_{w,s}$, the expression $F(t_1, \ldots, t_n)$ is a term of sort s.

Continuing with our running example, let $x : \mathsf{elem}$ and $s : \mathsf{stack}$ be two variables. Both

$$\begin{aligned} & \mathsf{push}(x, s) : \mathsf{stack} \\ \text{and} \quad & \mathsf{push}(x, \mathsf{push}(x, s)) : \mathsf{stack} \end{aligned}$$

are valid terms of sort stack, but $\mathsf{pop}(x)$ is not, for example, since x has the wrong sort.

All that remains is to define the syntax and semantics of first-order sentences. The definition we provide here is deliberately simplified since we will be able to give a much more precise account in SECTION 3.6.

2.4.8 *Definition* [ST11]. Let $\Sigma = \langle S, \mathcal{F}, \mathcal{P} \rangle$ be a signature. The *sentences* of first-order logic are built from the logical symbols $=$, $\to$, $\neg$, $\wedge$, $\vee$, $\forall$, $\exists$. The atomic sentences are

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

The sentences in general are defined inductively as follows:

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

We can now write sentences like $\forall x.\ \forall s.\ \mathsf{pop}(\mathsf{push}(x, s)) = s$. We'll defer an account of the semantics until SECTION 3.6.

But we are not quite done. There is one bit of structure we have failed to account for, and which characterises the institutional approach—the *signature morphism*.

2.4.9 *Definition* [ST11]. Let $\Sigma = \langle S, \mathscr{F}, \mathscr{P} \rangle$ and $\Sigma' = \langle S', \mathscr{F}', \mathscr{P}' \rangle$ be two signatures. A *signature morphism* $\sigma : \Sigma \to \Sigma'$ consists of a function $\sigma_{sorts} : S \to S'$, which will usually be written σ, as well as a pair of functions

$$\sigma_{funcs} : \prod_{w,s} \mathscr{F}_{w,s} \to \mathscr{F}'_{\sigma(w),\sigma(s)}$$
$$\sigma_{preds} : \prod_{w} \mathscr{P}_{w} \to \mathscr{P}'_{\sigma(w)}$$

respectively mapping sorts, function symbols, and predicate symbols, in such a way that the sorts are translated consistently with σ_{sorts}. We define $\sigma(w)$ as the action of σ_{sorts} on each of the sorts in w. In general we will write σ for any of the three components of a signature morphism since it will not cause confusion.

The existence of signature morphisms obligates us to explain how terms, algebras, and sentences are accordingly translated. Term and sentence translations are intuitively straightforward—they are direct liftings of signature morphisms and preserve the term and sentence structures. It's not necessary to define them here; instead I recommend taking a look at LISTINGS 3.6.18 & 3.7.5 for term and sentence translations respectively. That leaves us to define reduct algebras.

2.4.10 *Definition* [ST11]. Let Σ and Σ' be signatures, let $\sigma : \Sigma \to \Sigma'$ be a signature morphism, and let A' be a Σ'-algebra. The *reduct algebra* $A'|_\sigma$ is a Σ-algebra defined at each component of the algebra to be $A' \circ \sigma$.

This definition is worth explaining. Algebras (DEFINITION 2.4.4) are best thought of as functions providing a denotation for the symbols in a signature—as functions from symbols to 'real' mathematical objects.

In the presence of a change in signature $\sigma : \Sigma \to \Sigma'$, a Σ'-algebra can interpret symbols in Σ by first applying σ and interpreting the resulting Σ'-symbol; hence we 'precompose' A' by σ to obtain a Σ-algebra. Note that the direction is reversed; signature morphisms $\sigma : \Sigma \to \Sigma'$ map Σ'-algebras to Σ-algebras. Now is a good time to note the contravariance of the model functor in the definition of an institution: if $\sigma : \Sigma \to \Sigma'$ then $\mathsf{Mod}(\sigma) : \mathsf{Mod}(\Sigma') \to \mathsf{Mod}(\Sigma)$.

By this point, we deserve a more interesting example. Let's consider the relationship between boolean logic and what we'll call 'NAND logic', in which the only boolean operator is $\overline{\wedge}$, defined as $\lambda x, y.\ \neg(x \wedge y)$.

Let $\mathsf{boolsig}$ be a signature for boolean logic containing a single sort bool and the usual boolean data and functions—two constant symbols bT (true) and bF (false), and three function symbols $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT). Let $\mathsf{nandsig}$ be a signature with a single sort nbool, two constant symbols nT (true) and nF (false), and a function symbol $\overline{\wedge}$ (NAND). Suppose further that we have an infinite pool of variables $x, y, z, \ldots$ usable in either signature.

It's well known that the NAND operator is 'functionally complete', meaning that it can simulate all the other boolean operators. This fact is generally expressed as a list of equations.

- $\neg x = x \mathbin{\overline{\wedge}} x$
- $x \wedge y = (x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y)$
- $x \vee y = (x \mathbin{\overline{\wedge}} x) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} y)$

If we first construct a NAND algebra A with the standard semantics, then a boolean algebra can be given as the reduct $A|_\sigma$ of that algebra along the signature morphism σ indicated by the list of equations above. Let's describe this in more detail.

First, note that any signature Σ determines a *derived signature* [ST11, Definition 1.5.13] $\mathrm{TS}(\Sigma)$ in which the function symbols are replaced with terms, and in which the 'arity' of a term is given by the sorts of the free variables in that term. For example, $x \mathbin{\overline{\wedge}} y$ can be thought of as a function symbol in the derived signature with arity $[\mathsf{nbool}, \mathsf{nbool}]$,

the sorts of x and y. We can hence construct a signature morphism $\sigma : \mathsf{boolsig} \to \mathrm{TS}(\mathsf{nandsig})$ like so:

$$\begin{aligned}
\sigma_{sorts} &= \{\mathsf{bool} \mapsto \mathsf{nbool}\} \\
\sigma_{funcs} &= \{\mathsf{bT} \mapsto \mathsf{nT}, \\
&\qquad \mathsf{bF} \mapsto \mathsf{nF}, \\
&\qquad \neg \mapsto x \mathbin{\overline{\wedge}} x, \\
&\qquad \wedge \mapsto (x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y), \\
&\qquad \vee \mapsto (x \mathbin{\overline{\wedge}} x) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} y)\} \\
\sigma_{preds} &= \emptyset
\end{aligned}$$

Furthermore, since a derived signature $\mathrm{TS}(\Sigma)$ can be interpreted by a Σ-algebra by simply evaluating the term (look ahead to LISTING 3.6.11 for the details in Coq, but they are exactly as expected), the reduct $A|_\sigma$ is precisely a $\mathsf{boolsig}$-algebra with the right semantics—as long as we believe in the correctness of the translation. For example, here is the rough idea of how $\neg\mathsf{bT}$ would be interpreted by such a reduct:

$$(A|_\sigma)(\neg\mathsf{bT}) = A(\mathsf{nT} \mathbin{\overline{\wedge}} \mathsf{nT}) = \mathit{true} \mathbin{\overline{\wedge}} \mathit{true} = \mathit{false}$$

This example, and all the other examples we have seen so far, will be made precise in Coq in SECTION 3.6.5.

It's possible to give quite a bit more detail about everything so far introduced. Any such relevant details will appear later in their proper context. The concepts we have introduced so far are sufficient to define the institution *FOPEQ* for first-order predicate logic with equality. The next chapter is dedicated to constructing this object in Coq.

THREE

An Institution for First-Order Logic

In this chapter we will encode the institution for first-order logic outlined in CHAPTER 2 in Coq and prove its satisfaction condition. We'll cover some common constructions in the formalisation and familiarise ourselves with how to read mathematics and proofs in Coq.

We've formalised all the work presented here in Coq.[11] We iterate on a formalisation of universal algebra in Agda [GGP18] and depend on a formalisation of category theory by John Wiegley [Wie14]. The work in this chapter first appeared in a much abridged form in [RM22].

3.1 *The Coq Proof Assistant*

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 elegant representations of complex mathematical objects such as those found in category theory, institution theory, and universal algebra. Coq is a popular choice for studying the semantics of programming languages. Adam Chlipala's *Certified Programming with Dependent Types* [Chl13] and the *Software Foundations* series, particularly volumes one and two [Pie+23a; Pie+23b], are textbooks containing material specifically in aid of studying the foundations of programming languages in Coq. The CompCert compiler for a subset of C is developed and verified in Coq [Ler09], and such work requires representing the syntax and semantics of C in Coq. What we are doing

here is not unlike formalising programming languages—institutions can represent programming language semantics directly such as the institutions *FPL* or *IMP* (both from [ST11, §4.1] and respectively denoting simple functional and imperative languages) and indeed the institution we will define later in SECTION 5.3.

We will assume some basic familiarity with Coq syntax for the remainder of this thesis. The notation we use should be readable even to those unfamiliar with Coq. Nevertheless, we'll discuss some common constructions used in proofs throughout the thesis. Be sure to consult the tactic index on page 135 if you are unfamiliar with Coq's tactic language.

If you would like to follow along with some of the simpler examples that do not depend on our developments but would rather not install Coq, I recommend jsCoq.

https://coq.vercel.app/scratchpad.html

I can also recommend Adam Chlipala's *Certified Programming with Dependent Types* [Chl13] as a good, fast introduction to Coq, especially since many of the techniques explained there appear in this thesis.

3.1.1 *Mathematical notation in Coq*

Type theoretic concepts are represented differently in Coq compared to standard mathematical writing. Dependent products or functions are represented by $\forall x : X.\ P(x)$ in Coq but by

$$\prod_{x:X} P(x)$$

in mathematical notation. Similarly, dependent sums are represented by $\exists x : X.\ P(x)$ in Coq but by

$$\sum_{x:X} P(x)$$

in mathematical notation. This means that one should read $\forall$ in Coq notation as describing a dependent function and not as literally meaning 'for all' in the common logical sense; and similarly that $\exists$ refers to a

dependent sum and not just 'there exists'. This is listed in the notation index on page 133.

3.1.2 *Common constructions and their meanings in Coq*

Rewriting with a propositional equality Recall that we discussed in SECTION 2.2.2 the distinction between internal and external notions of equality in type theory, with the internal notion referred to as 'propositional equality'. We commonly want to use an equality $p : x = y$ to rewrite a term involving x into a term involving y, and Coq's tactic system makes this simple and intuitive. Consider the following illustrative example, proving for all $n : \mathbb{N}$ that $n = 3$ implies $n + 1 = 4$.

```
Proposition silly (n : nat) :                                    3.1.1
  n = 3 -> n + 1 = 4.
Proof.
  intros p.  (* p : n = 3 *)
  rewrite p.  (* n + 1 = 4  ⇝  3 + 1 = 4 *)
  reflexivity.  (* equal by computation *)
Defined.
```

Recall that we can only prove $a = b$ in type theory by constructing a proof term with the type $a = b$. This implies that Coq's tactic language is explicitly constructing such a term. Because we have closed the proof with `Defined`, we can see the proof term Coq constructed.

```
silly =                                                          3.1.2
  λ (n : nat) (p : n = 3),
  eq_ind 3 (λ k : nat, k + 1 = 4) eq_refl n p^
```

Here eq_ind takes five arguments.* The first and fourth can be ignored since they are the left- and right-hand side respectively of the equality $p^{-1} : 3 = n$. The second argument is the type family $P(k) \triangleq k+1 = 4$, which points to the 'subject' of the rewrite, and the final argument is the proof term p^{-1} that we are using to rewrite the goal. The third argument,

*eq_ind and eq_rect are the same thing—the only difference is that the type family argument must have the type $A \to$ Prop rather than $A \to$ Type.

eq_refl, is the term whose type we are rewriting. Note that here eq_refl actually has the type $3 + 1 = 4$—and the reason it may have that type is because $3 + 1$ and 4 are judgementally equal. So the term above may be rewritten in our notation as

$$\mathsf{rew}\, p^{-1} \,\mathsf{in}\, (\mathsf{eq\_refl} : 3 + 1 = 4) : n + 1 = 4$$

In sum, the proof term says: Let $n : \mathbb{N}$ and $p : n = 3$. We know that $3 + 1 = 4$ by reflexivity (i.e. $\mathsf{eq\_refl} : 3 + 1 = 4$), and hence we may construct $\mathsf{rew}\, p^{-1} \,\mathsf{in}\, (\mathsf{eq\_refl} : 3 + 1 = 4) : n + 1 = 4$. Since we have constructed a term of type $n + 1 = 4$, we're done.

Rewrites always construct proof objects of the form $\mathsf{rew}\, p \,\mathsf{in}\, t$. Sometimes these rewrites appear in unexpected places. Let's consider a typical such example. Later on we will define *first-order terms* in Coq; such terms will have a type of the form $\mathsf{Term}(A, \Gamma)_s$, where Γ is a list of sorts. We will define a term translation which lifts a signature morphism to the level of terms. This transformation, specialised to the identity signature morphism, will 'do nothing', as expected. But the type of the resulting term will be $\mathsf{Term}(A, \mathsf{map}\ \mathsf{id}\ \Gamma)_s$ and not $\mathsf{Term}(A, \Gamma)_s$. Since these types are not judgementally equal, we cannot even state the equality $\mathsf{id}(t) = t$. But since the types are *propositionally* equal, we may instead write

$$\mathsf{id}(t) = \mathsf{rew}\, [\mathsf{Term}(A, -)_s]\, p \,\mathsf{in}\, t$$

where $p : \Gamma = \mathsf{map}\ \mathsf{id}\ \Gamma$. This can now be proved normally, and we will do so in SECTION 3.7.

Constructing terms with tactics We just saw that Coq's tactic language constructs proof terms that would otherwise be tedious to write out ourselves. But since there is no difference in Coq between a proof and any other value, we can also use Coq's tactic language to construct any value at all. This is sometimes more convenient, especially when the types of values become complicated, or when it's not known what details should be supplied yet.

Let's consider the example of constructing dependent records. Dependent records are n-tuples where subsequent terms may depend on

previous ones. There is not really any such distinction in written mathematics between 'dependent' and 'non-dependent' tuples, so it may be best to think of them as tuples with named fields. They are denoted like so—

```
{|                                                                    3.1.3
  field1 := value1 ;
  field2 := value2 ;
|}
```

We will commonly construct dependent records via Coq's tactic language rather than providing the terms directly. To do this, we can write—

```
unshelve refine {|                                                    3.1.4
  field1 := _ ;
  field2 := _ ;
|}.
- (* construct value1 *)
- (* construct value2 *)
```

Another common (and more concise) way to do this is to simply write—

```
unshelve esplit.                                                      3.1.5
- (* construct value1 *)
- (* construct value2 *)
```

This is useful when we do not wish to name the fields, but has the minor disadvantage of hiding the value which is being constructed from the reader.

3.2 *Representing Institutions*

An institution can be described directly as a dependent record in Coq. The reader should compare the following with the mathematical definition, DEFINITION 2.3.1.

```
Class Institution := {                                                3.2.1
  Sig : Category ;
  Sen : Sig ⟶ SetCat ;
```

```
  Mod : Sig^op ⟶ Cat ;

  interp : ∀ (Σ : Sig), Mod Σ -> Sen Σ -> Prop
    where "M ⊨ φ" := (interp _ M φ) ;

  sat : ∀ (Σ Σ' : Sig) (σ : Σ ~> Σ') (φ : Sen Σ) (M' : Mod Σ'),
    M' ⊨ fmap[Sen] σ φ <-> fmap[Mod] σ M' ⊨ φ
}.
```

The category-theoretic definitions and notations used above come from John Wiegley's category theory library [Wie14]. The notation should be self-explanatory. The institution can often be inferred from context, but for situations where it needs to be made explicit, we use the following notations—

```
Notation "Sig[ I ]" := (Sig (Institution := I)).                    3.2.2
Notation "Sen[ I ]" := (Sen (Institution := I)).
Notation "Mod[ I ]" := (Mod (Institution := I)).
Notation "sat[ I ]" := (sat (Institution := I)).
```

Before we begin constructing *FOPEQ*, we need to explain three crucial concepts and their role in the formalisation: indexed types, heterogeneous lists, and type theory's identity type.

3.3 *Indexed Types*

We presented value-indexed sets in SECTION 2.4. Indexed types in Coq look similar and are represented by a type family $J \to \mathcal{U}$, where J is the index type and $\mathcal{U}$ is a universe of types.

A good example of an indexed type is a *vector*. Vectors are like lists, but the length of the list is visible at the type level.

```
Inductive Vec A : nat -> Set :=                                     3.3.1
| VNil : Vec A 0
| VCons : ∀ n, A -> Vec A n -> Vec A (S n).
```

This defines a family of types—one for each natural number. The term $v : \mathsf{Vec}(\mathsf{nat})_3$, for instance, is a vector of length 3 containing exactly

three values of type nat. The constructors for this type guarantee that any constructed vector with type $\mathsf{Vec}(A)_n$ has length n.

There is another way to encode this type which 'demotes' the index to the value level.

```
Definition Vec A := { x : list A * nat | len (fst p) = snd p }.   3.3.2
```

Let's call this the non-indexed representation. To construct a value of $\mathsf{Vec}(A)$, we construct a pair $x = (\ell, n)$ where ℓ is a list and n is a natural number denoting its length. The list need not actually have that length, so we must also provide a proof $p : \mathsf{len}\ \ell = n$. Carrying this proof around is the cost of moving to a non-indexed representation.

There are advantages and disadvantages to each representation, which we need not get into for the purposes of this discussion. The important point is this: When we moved from an indexed to non-indexed representation, the information which was encoded at the type level became something like a well-definedness proof at the value level.

But sometimes moving to a non-indexed representation does not obligate us to carry any proofs around. Consider the function symbol type $\mathcal{F} : \mathsf{List}(S) \to S \to \mathcal{U}$. Normally we'd write $F \in \mathcal{F}_{w,s}$ to mean that F has arity w and result sort s. But we can instead write $F \in \mathcal{F}$ and insist that $\mathcal{F}$ come with an associated function $\mathsf{funsig} : \mathcal{F} \to \mathsf{List}(S) \times S$. This representation suffers from none of the issues of the non-indexed vector type since it is impossible for the value and its index to 'disagree', and since they can't disagree we do not need to carry around a proof that they agree. One simply *declares* a function symbol with a given name and type. Note that in the formalisation we will split funsig into two functions $\mathsf{ar} : \mathcal{F} \to \mathsf{List}(S)$ and $\mathsf{res} : \mathcal{F} \to S$, respectively mapping function symbols to their arity and result sort.

Let's describe the situation more generally in set-theoretic terms. An indexed family of sets is a function $S : J \to \mathsf{Set}$, and so to obtain an 'element' of such a family, we must first have in mind an index $j \in J$. The informal '$x \in S$' means that there is some $j \in J$ such that $x \in S_j$. The dependency is important—we need to know the index before we can

even denote the set S_j containing x. This dependency can be reversed by instead providing a set

$$\mathcal{S} = \bigcup_{j \in J} \{ (x, j) \mid x \in S_j \} \tag{3.1}$$

and a function $f : \mathcal{S} \to J$ defined by $(x, j) \mapsto j$.

This alternative encoding of function and predicate symbols is mentioned in [ST11, §1.2], but it is not pursued further. We call such data 'tagged' and represent it as follows.

```
Record Tagged T := {                                    3.3.3
  tagged_data :> Type ;
  get_tag :> tagged_data -> T ;
}.
```

This type exactly appears in, for example, Andrej Bauer, Philipp G. Haselwarter and Peter LeFanu Lumsdaine's work on a general representation of dependent type theories in Coq [BHL20], but they use it to represent indexed families in the traditional sense. We reverse the position of data and index.

For many data types, we use the representation $A : \mathsf{Tagged}(J)$ in place of the indexed type $A : J \to \mathcal{U}$, demoting the index to the value level. For function symbols, the symbols themselves are contained in tagged_data and the ar and res functions are combined in get_tag. Since it will not introduce confusion, we will continue to use the notation $F \in \mathcal{F}_{w,s}$ to denote a function symbol with arity w and result sort s.

There are trade-offs associated with this choice of encoding. For us, it crucially enables a significantly more straightforward description of sums and products of signatures, which will be important for constructing pushouts of signatures and for eventually proving the amalgamation property for key institutions like *FOPEQ* and *EVT*. (We explain pushouts and amalgamation later on in SECTION 6.4.) But one big drawback is that we lose some flexibility. Terms in a given signature can be thought of as 'function symbols' with 'arities' given by the variable context. This allows us to regard a term such as $\mathsf{push}(x, y)$ as a function symbol roughly

equivalent to $\lambda x, y.\ \mathsf{push}(x, y)$. We get this for free with indexed types but we need to explicitly perform the transformation indicated in (3.1) to get it working with the tagged representation.

To further motivate this type, we will discuss *tagged morphisms*. Morphisms of tagged data are just regular functions on the data, but that function must be consistent with the ambient morphism on tags. More specifically, given a translation of tags t, a tagged morphism $f : X \xrightarrow{t} Y$ is constrained to translate the data such that $\mathsf{tag}(f(x)) = t(\mathsf{tag}(x))$. This condition, specialised to function symbols, is precisely the condition that a morphism of function symbols should translate the function's arity and result sort consistently with a morphism on sorts. Tagged morphisms are defined in Coq as follows.

```
Definition tagged_morphism                                          3.3.4
    [A B : Type] (t : A -> B) (X : Tagged A) (Y : Tagged B) :=
  { f : X -> Y | ∀ x : X, get_tag (f x) = t (get_tag x) }.
```

The condition on a tagged morphism $f : X \xrightarrow{t} Y$ is a sort of commutativity lemma. We will give the lemma an explicit name so that it is easy to apply in later contexts. (The function proj2_sig gets the second component of a dependent pair.)

```
Definition tagged_morphism_commutes                                 3.3.5
    [A B : Type] [t : A -> B] [X Y]
    : ∀ (f : tagged_morphism t X Y) (i : X),
        Y (f i) = t (X i) :=
  @proj2_sig _ _.
```

There are lots of useful things we can prove about this type. First, we will introduce what we will from now on refer to as an *equality lemma*—a list of explicit conditions for proving equality of two objects of a particular type. Tagged morphisms consist of a function and a coherence proof—and therefore by proof irrelevance (recall the discussion in SECTION 2.2.3) they are equal if the functions are equal. We encode this reasoning explicitly as follows.

```
Lemma tagged_morphism_eq {T1 T2 X Y} {t : T1 -> T2}                3.3.6
  (f g : tagged_morphism t X Y)
  : proj1_sig f = proj1_sig g -> f = g.
Proof.
  intros H. destruct f, g.
  apply subset_eq_compat, H.
Qed.
```

We will apply this lemma whenever our goal is an equality of two tagged morphisms.

We also define composition for tagged morphisms, with the proof lifted right out of [BHL20].

```
Definition tagged_morphism_compose                                 3.3.7
  [A B C : Type] [gᵢ : B -> C] [fᵢ : A -> B] [X Y Z]
  (g : tagged_morphism gᵢ Y Z) (f : tagged_morphism fᵢ X Y)
  : tagged_morphism (gᵢ ∘ fᵢ) X Z.
Proof.
  exists (g ∘ f).
  intros. refine (eq_trans _ _).
  * apply tagged_morphism_commutes.
  * apply f_equal. apply tagged_morphism_commutes.
Defined.
```

Though this appears to be an odd way to prove this lemma, it is important that the proof term is constructed this way—presuming we do not eliminate it with proof irrelevance. The proof term looks like this:

```
(λ x : X,                                                          3.3.8
  eq_trans (tagged_morphism_commutes g (f x))
    (f_equal gᵢ (tagged_morphism_commutes f x)))
```

Later on in SECTION 3.5 we will discuss why we might want to construct proof terms in such a manner.

3.4 *Heterogeneous Lists*

Suppose we have at hand a function $A : S \to \mathcal{U}$ interpreting sorts as Coq types and a function symbol $F : w \to s$, where w is a list of sorts

and s is a sort. We need to construct the type

$$A_{w_1} \to A_{w_2} \to \cdots \to A_{w_n} \to A_s \tag{3.2}$$

or something equivalent. A type such as this can be defined directly by case analysis on w.

```
Fixpoint Curried {J : Type} (A : J -> Type) w s : Type :=          3.4.1
  match w with
  | [] => A s
  | j :: js => A j -> Curried A js s
  end.
```

But this type synonym is surprisingly cumbersome to work with, for reasons we will indicate in SECTION 3.6.3. We need a different approach.

Gunther *et al.* in their paper formalising universal algebra in Agda [GGP18] identified heterogeneous lists as a promising candidate. A similar definition to Gunther's exists in *Certified Programming with Dependent Types* by Adam Chlipala [Chl13].

What is a heterogeneous list? Lists are normally homogeneous insofar as they contain elements of the same type—as in DEFINITION 2.4.2. By contrast, *heterogeneous lists*—henceforth *h-lists*—can contain elements of different types. There are lots of simple ways to define h-lists, but they are often not useful. For example, the heterogeneity requirement is easy to meet by simply hiding type information, but we need more control—we need to know exactly the type of each of the elements of the h-list. Here is the definition in Coq.

```
Context (J : Type) (A : J -> Type).                                3.4.2

Inductive HList : list J -> Type :=
| HNil  : HList []
| HCons : ∀ {j js}, A j -> HList js -> HList (j :: js).

Inductive member (j : J) : list J -> Type :=
| HZ : ∀ {js}, member j (j :: js)
| HS : ∀ {j' js}, member j js -> member j (j' :: js).
```

The h-list type comes with another inductive type member which can be thought of as a 'typed' natural number. A 'number' n : member $x\ \ell$ can be thought of as a constructive proof that x appears in the list ℓ at index n. It's used to index h-lists in the following way:

```
Equations nth [j js] (v : HList A js) : member j js -> A j :=      3.4.3
  nth (HCons x _ ) HZ     := x ;
  nth (HCons _ xs) (HS m) := nth xs m .
```

The coq-equations plugin [Soz10] provides the 'equations' syntax above—it can handle many of the difficulties of dependent pattern matching for us. We will use the following notations for h-lists:

```
Infix ":::" := HCons : hlist_scope.                                3.4.4
Notation "⟨ ⟩" := HNil : hlist_scope.
Notation "⟨ x ⟩" := (HCons x HNil) : hlist_scope.
Notation "⟨ x ; y ; .. ; z ⟩" :=
  (HCons x (HCons y .. (HCons z HNil) ..)) : hlist_scope.
```

The point of h-lists is that they allow us to construct the type

$$\sum_{i \in w} A_i \to A_s \tag{3.3}$$

which is an uncurried form of (3.2). In fact, (3.2) and (3.3) are completely interconvertible via the following generic curry and uncurry functions.

```
Equations curry {J : Type} {A : J -> Type} {w s}                   3.4.5
                (F : HList A w -> A s) : Curried A w s :=
  curry (w := [])   F := F ⟨⟩ ;
  curry (w := _::_) F := λ x, curry (λ xs, F (x ::: xs)) .

Equations uncurry {J : Type} {A : J -> Type} {w s}
                  (F : Curried A w s) : HList A w -> A s :=
  uncurry (w := [])   F ⟨⟩ := F ;
  uncurry (w := _::_) F (x ::: xs) := uncurry (F x) xs .
```

As an example, let's consider what HList values look like. Recall the running example stack_sig we introduced in SECTION 2.4. Let $J =$

$\{\mathsf{elem}, \mathsf{stack}\}$, let $w = [\mathsf{elem}, \mathsf{stack}, \mathsf{elem}]$, and let $A : J \to \mathcal{U}$ be a function interpreting the sorts as types, defined by $\mathsf{elem} \mapsto \mathbb{N}$ and $\mathsf{stack} \mapsto \mathsf{List}(\mathbb{N})$. Then $\langle 2, [3, 4], 0\rangle$ would be a valid term of type $\mathsf{HList}(A, w)$.

Operations on h-lists are similar to those for lists. A full list can be seen on the GitHub repository at Core/HList.v, but we'll explicitly mention the most common one: *reindexing*. The idea is that we have the following equivalence of types

$$\mathsf{HList}(A \circ f, w) \cong \mathsf{HList}(A, \mathsf{map}\ f\ w)$$

and we can define two functions that will convert these types.

```
Fixpoint reindex                                                    3.4.6
    {J K : Type} (f : J -> K)
    {A : K -> Type} {js : list J} (v : HList (A ∘ f) js)
    : HList A (map f js) :=
  match v with
  | ⟨⟩ => ⟨⟩
  | a ::: v' => a ::: reindex f v'
  end.

Equations reindex'
    {J K : Type} (f : J -> K)
    {A : K -> Type} {w : list J} (v : HList A (map f w))
    : HList (A ∘ f) w :=
  reindex' f (w := []) ⟨⟩ := ⟨⟩ ;
  reindex' f (w := _::_) (x ::: xs) :=
    x ::: reindex' f xs .
```

Reindexing also has the following important relationship to hmap (which is the equivalent of map for h-lists) which turns up later in the proof of LISTING 3.7.24.

```
Lemma hmap_reindex [A B : K -> Type]                                3.4.7
  (f : J -> K) (g : ∀ k, A k -> B k) `(v : HList (A ∘ f) w)
  : hmap g (reindex f v) = reindex f (hmap (g ∘ f) v).
```

3.5 *A Closer Look at the Identity Type*

Martin Hoffmann and Thomas Streicher in [HS98] suggest that the identity type has the structure of a *groupoid*. This structure turns out to be very useful for us.

3.5.1 *Definition.* A *groupoid* is a category in which every morphism is an isomorphism—that is, for any morphism $f : a \to b$ there exists a morphism $g : b \to a$ such that $f \circ g = \mathsf{id}_a$ and $g \circ f = \mathsf{id}_b$. A *groupoid morphism* is a functor between two groupoids.

The analogy here is to think of identities $p : a = b$ as arrows $a \to b$ in the groupoid. We won't be very detailed or precise about this analogy—see [UFP14, §2.1] in particular for more details. We will focus only on the salient points. We will also indicate the names for these constructions in Coq, referring back to this section later when necessary.

All morphisms are isomorphisms Each proof $p : a = b$ corresponds to a proof $p^{-1} : b = a$, the symmetric proof. This is eq_sym p in Coq, notated p^.

Morphisms can be composed Two proofs $p : a = b$ and $q : b = c$ can be 'concatenated' to form the transitive proof $p \cdot q : a = c$. This is eq_trans p q in Coq.

Functions are functors Any function $f : A \to B$ behaves functorially on identity proofs. That is to say, there is an operation ap_f taking $p : a = b$ to $\mathsf{ap}_f(p) : f(a) = f(b)$. This encodes the reasoning that functions respect equality and is denoted f_equal f p in Coq.

Type families are fibrations If $P : A \to \mathcal{U}$ is a fibration then—by analogy to classical homotopy theory—it induces a 'path lifting' operation, mapping each morphism $p : a = b$ to a morphism

$$\mathsf{transport}^P(p, -) : P(a) \to P(b)$$

We've seen this already as the property of *indiscernibility of identicals* in SECTION 2.2. This is written eq_rect a b p in Coq with the shorthand

rew p in $(-)$. We will occasionally use the notation $p_* : P(a) \to P(b)$ outside of a Coq context when the type family is clear.

By virtue of these observations, the identity type satisfies many useful properties. Simple examples include $p \cdot p^{-1} = \mathsf{refl}$, but let's also list some more complicated ones.

3.5.2 *Lemma* [UFP14]. Let $P, Q : A \to \mathcal{U}$ be type families, let $x, y : A$ and $p : x = y$, let $f : \prod_{(a:A)} P(a) \to Q(a)$ be a dependent function, and let $t : P(x)$. Then

$$\mathsf{transport}^{Q}(p, f_x(t)) = f_y(\mathsf{transport}^{P}(p, t))$$

3.5.3 *Lemma* [UFP14]. Let $P : B \to \mathcal{U}$ be a type family, let $f : A \to B$ be a function, let $x, y : A$ and $p : x = y$, and let $t : P(f(x))$. Then

$$\mathsf{transport}^{P \circ f}(p, t) = \mathsf{transport}(\mathsf{ap}_f(p), t)$$

3.5.4 *Lemma* [UFP14]. Let $f : A \to B$ and $p : x = y$. Then

$$\mathsf{ap}_f(p^{-1}) = (\mathsf{ap}_f(p))^{-1}$$

Now recall that eq_rect computes in the sense indicated by (2.1) in SECTION 2.2. If we can simplify using any one of the facts above (or the many more, since this is just a sample) then we can make progress in proofs involving the explicit manipulation of identity proofs. This will become relevant to us in SECTION 3.7, specifically in LISTING 3.7.10.

3.6 *First-Order Logic in Coq*

This section details the translation of the mathematical concepts defined in SECTION 2.4 into Coq. These translations are occasionally nontrivial. Keep in mind the notes in SECTION 2.2, which we will reference as needed.

3.6.1 *First-order signatures*

First-order signatures (DEFINITION 2.4.3) are represented by a record, mirroring the mathematical definition exactly, and making use of the tagged data we defined in SECTION 3.3.

```
Record Signature := {                                              3.6.1
  Sorts : Type ;
  Funcs : Tagged (list Sorts * Sorts) ;
  Preds : Tagged (list Sorts) ;
}.
```

As promised in SECTION 1.3, and giving some context to the discussion in SECTION 2.2, we will quickly discuss some issues of encoding now that they are relevant. We have represented a set of sorts as a type—any type at all. The set-theoretic definition of first-order signatures given in DEFINITION 2.4.3 presumes only that the sorts form a set and do not impose any further conditions. In practice a set of sorts is usually finite but the definition does not judge.

It is unclear which definition is more permissive—though it is hard to see how either could be substantially more permissive than 'any type' and 'any set'. The correspondence here is not one-to-one, because types do not correspond one-to-one with sets, but it is close—sets are the basic objects of set theory, and types are (one of) the basic objects of type theory; and indeed, types are meant to capture a specific sense of set in a manner claimed to be closer to what is informally meant in mathematical practice. (Again, see SECTION 2.2 and [Shu13] for a more detailed discussion about this.)

Let's continue the encoding with the example signature stack_sig from SECTION 2.4. First, write out the basic symbols—

```
Inductive stack_sorts := elem | stack.                             3.6.2
Inductive stack_funcs_names := empty | push | pop.
Inductive stack_preds_names := is_empty.
```

and then give each function and predicate symbol a type.

```
Definition stack_funcs := {|                                    3.6.3
  tagged_data := stack_funcs_names ;
  get_tag F :=
    match F with
    | empty => ([], stack)
    | push => ([elem; stack], stack)
    | pop => ([stack], stack)
    end ;
|}.

Definition stack_preds := {|
  tagged_data := stack_preds_names ;
  get_tag P :=
    match P with
    | is_empty => [stack]
    end ;
|}.
```

This data can be packed into a signature.

```
Definition stack_sig : Signature := {|                          3.6.4
  Sorts := stack_sorts ;
  Funcs := stack_funcs ;
  Preds := stack_preds ;
|}.
```

3.6.2 *First-order models*

First-order models, also known as *algebras* (DEFINITION 2.4.4), need to interpret the sorts of a signature as Coq types and the function and predicate symbols as Coq functions and predicates with the right types. Recall the h-lists we defined in LISTING 3.4.2.

```
Record Algebra Σ := {                                           3.6.5
  interp_sorts :> Sorts Σ -> Type ;
  interp_funcs F : HList interp_sorts (ar F) -> interp_sorts (res F) ;
  interp_preds P : HList interp_sorts (arP P) -> Prop
}.
```

Algebras are regarded as mapping symbols in the signature to objects in the real mathematical universe. Since that universe is in our case Coq's type theory, then they must map sorts to type-theoretic types, function symbols to Coq functions between those types, and predicate symbols to Coq predicates, i.e. Coq functions with codomain Prop.

Let's define the algebra for stack_sig outlined in SECTION 2.4. It turns out the curried representation outlined in SECTION 3.4 is useful for constructing concrete algebras in a readable way since it allows us to directly use standard library functions from Coq—though we must be explicit about the types since Coq cannot infer them for us.

```
Definition stack_nat_is (s : Sorts stack_sig) : Type :=          3.6.6
  match s with
  | elem  => nat
  | stack => list nat
  end.

Local Notation stack_fn := (Curried stack_nat_is) (only parsing).
Local Notation stack_args := (HList stack_nat_is) (only parsing).

Definition stack_nat_if (F : Funcs stack_sig)
    : stack_args (ar F) -> stack_nat_is (res F) :=
  uncurry match F with
  | push  => List.cons    : stack_fn [elem; stack] stack
  | pop   => @List.tl nat : stack_fn [stack] stack
  | empty => List.nil     : stack_fn [] stack
  end.
```

We get no such niceties for predicate symbols, though the definition of new predicates directly with coq-equations is usually straightforward.

```
Equations stack_nat_is_empty                                     3.6.7
    (args : stack_args [stack]) : Prop :=
  stack_nat_is_empty ( s ) := s = [].

Definition stack_nat_ip (P : Preds stack_sig)
    : HList stack_nat_is (Preds stack_sig P) -> Prop :=
  match P with
```

```
| is_empty => stack_nat_is_empty
end.
```

3.6.3 *First-order terms*

There are so far only minor differences from Gunther's work in Agda [GGP18]. The first major deviation is in the definition of variables (DEFINITION 2.4.6) and terms (DEFINITION 2.4.7), which we gather into a single inductive type.

```
Inductive Term Σ Γ : Sorts Σ -> Type :=                                    3.6.8
| var s : member s Γ -> Term s
| term F : HList Term (ar F) -> Term (res F).
```

Recall that we defined a term of sort s as either a variable $x : s$ or as a function symbol $F \in \mathcal{F}_{w,s}$ 'applied' to a list of terms $t_1 : w_1, \dots, t_n : w_n$, written $F(t_1, \dots, t_n)$. But variables are best represented not by members of an indexed set, which can be difficult to control, but instead by what we might call 'dependent de Bruijn indices'. Chlipala discusses this idea in [Chl13, Chapter 9]. The reasons will be discussed in their proper place in SECTION 3.6.4 when we encode quantifiers. We will also defer examples of first-order terms until then because they are best understood in that context.

Note that the difficulties of the Curried representation are now much more apparent—term F could not have the type

$$\mathsf{Curried}\ \mathsf{Term}\ (\mathsf{ar}\ F)\ (\mathsf{res}\ F)$$

since, in the first place, constructors cannot have a variable number of arguments, but also because Coq has no mechanism by which to conclude that the type of the fully applied constructor is $\mathsf{Term}\ (\mathsf{res}\ F)$.

3.6.4 *The semantics for first-order logic*

We can now start to build the syntactic and semantic structure of first-order sentences—see FOL/Sentence.v and Institutions/InsFOPEQ.v for the full details. The syntax is as follows.

```
Inductive FOL : list (Sorts Σ) -> Type :=                    3.6.9
| Forall Γ s : FOL (s :: Γ) -> FOL Γ
| Exists Γ s : FOL (s :: Γ) -> FOL Γ
| Equal Γ s : Term Σ Γ s -> Term Σ Γ s -> FOL Γ
| Pred Γ P : HList (Term Σ Γ) (arP P) -> FOL Γ
| ...
```

We omit the other connectives since their definitions are straightforward. The quantifiers, however, require some explanation.

Syntactically, a quantifier accepts as argument a sentence in which at least one variable may appear free and binds it. Recall that first-order variables were defined as typed de Bruijn indices into a list of sorts. This list of sorts is called a *context*. A context contains information about how many free variables there can be and what sorts they have. For example, if ψ is a sentence with context $s :: \Gamma$, then it may have at least one free variable at index 0 with the sort s—it may have more if Γ is non-empty. The variable need not actually appear in the sentence; the sentence True, for example, has no free variables, but can have a nonempty context. Quantifiers Q take sentences like ψ with a nonempty context $s :: \Gamma$ and construct a new sentence $\mathsf{Q}_s.\ \psi$ with context Γ (one fewer free variables). Syntactically, at least, this is what it means to bind a variable. Note that quantifiers always bind the first free variable listed, but this is not a serious limitation because the order in which the free variables are listed in the context does not matter. Formally, we have the following syntactic formation rule:

$$\frac{s :: \Gamma \vdash \phi}{\Gamma \vdash \mathsf{Q}_s.\ \phi}$$

To interpret a first-order sentence, we must decide what the logical symbols mean and what values the free variables will get. If θ is an 'environment' providing values for the variables in Γ, then we denote the semantic

interpretation of a sentence ϕ with free variables from Γ by an algebra A with environment θ by $A \models^{\theta} \phi$. Precisely, in the case of the quantifiers, we have

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

The idea then is to collect the values bound in Coq with the right types into a h-list so that, when evaluation of the sentence structure is finished and it comes time to interpret the variables, the values can be passed to the right locations. 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                                                    3.6.10
    (A : Algebra Σ) (φ : FOL Σ Γ) (θ : HList A Γ) : Prop :=
  match φ with
  | Forall s ψ => ∀ x : A s, interp_fol A ψ (x ::: θ)
  | Exists s ψ => ∃ x : A s, interp_fol A ψ (x ::: θ)
  | Equal u v  => eval_term A u θ = eval_term A v θ
  | Pred P ts  => interp_preds A P (map_eval_term A ts θ)
  | ...
  end
```

We have not defined the term evaluation function yet, so we will do so now. It is verbose but uncomplicated.

```
Equations eval_term (A : Algebra Σ) (t : Term Σ Γ s)                   3.6.11
    : HList A Γ -> A s := {
  eval_term _ (var i) θ := HList.nth θ i ;
  eval_term A (term F args) θ :=
    interp_funcs A F (map_eval_term A args θ)
} where map_eval_term (A : Algebra Σ) (args : HList (Term Σ Γ) w)
    : HList A Γ -> HList A w := {
  map_eval_term _ ⟨⟩ _ := ⟨⟩ ;
  map_eval_term A (t ::: ts) θ :=
    eval_term A t θ ::: map_eval_term A ts θ
}.
```

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.

The mutually-defined map_eval_term appears redundant. It should just be the result of mapping eval_term over the arguments—and indeed, we can prove it:

```
Lemma map_eval_term_hmap                                              3.6.12
  {Σ : Signature} (A : Algebra Σ) {Γ : Ctx Σ} {w : Arity Σ}
  (args : HList (Term Σ Γ) w) (env : HList (interp_sorts A) Γ) :
map_eval_term A args env = hmap (λ _ t, eval_term A t env) args.
```

The reason the mutual definition is necessary is because, without it, Coq cannot prove that any argument to eval_term structurally decreases. Since we have the proof in LISTING 3.6.12, little is lost. Other mutual definitions appearing in this thesis will have similar auxiliary lemmas, such as in LISTING 3.6.18.

Let's consider some an example to see how the pieces fit together. We will encode the first-order sentence $\forall x, s.\ \mathsf{pop}(\mathsf{push}(x, s)) = s$.

```
Local Notation stack_term := (term (Σ := stack_sig)).                 3.6.13
Local Notation stack_pred := (Pred (Σ := stack_sig)).

Local Notation x₁ := HZ.
Local Notation x₂ := (HS x₁).

Definition example_sentence₁ : FOL stack_sig [] :=
  Forall (Forall (
    Equal (stack_term pop ( stack_term push ( var x₂ ; var x₁ ) ))
          (var x₁))).
```

This works like a standard de Bruijn representation. To find which variable is bound by which quantifier, walk back through the representation and count the quantifiers until you reach the number equal to the variable. For example, x_2 above is bound by the first quantifier. The proof of this

follows trivially by computation—and it should, since by computation this sentence is equal to the Coq proposition $\forall x, s.\ \mathsf{tl}\ (x :: s) = s$.

```
Theorem example_correct₁ : stack_nat ⊨ example_sentence₁.        3.6.14
Proof. cbn; reflexivity. Qed.
```

A more detailed derivation would show the intermediate steps of this process, but the basic transformation works like this:

```
   stack_nat ⊨ example_sentence₁                                 3.6.15

↣  ∀ x s, interp_fol stack_nat
     (Equal (stack_term pop ⟨ stack_term push ⟨ var x₂; var x₁ ⟩ ⟩)
            (var x₁))
     ⟨ s; x ⟩

↣  ∀ x s, eval_term stack_nat
       (stack_term pop ⟨ stack_term push ⟨ var x₂; var x₁ ⟩ ⟩)
       ⟨ s; x ⟩
     = eval_term stack_nat (var x₁) ⟨ s; x ⟩

↣  ∀ x s, uncurry tl
           ⟨ uncurry cons ⟨ nth ⟨ s; x ⟩ x₂; nth ⟨ s; x ⟩ x₁ ⟩ ⟩
     = nth ⟨ s; x ⟩ x₁

↣  ∀ x s, tl (x :: s) = s
```

We have presented everything required so far besides signature morphisms and related constructions. This is where things get trickier.

3.6.5 *Signature morphisms, reduct algebras, term translations*

Signature morphisms (DEFINITION 2.4.9) are characteristic of the institutional approach. In Coq, they are represented by the tagged morphisms we introduced in SECTION 3.3.

```
Definition lift_ty [A B] (f : A -> B) (ℓ : list A * A) :=        3.6.16
  (map f (fst ℓ), f (snd ℓ)).

Record SigMorphism Σ Σ' := {
  on_sorts :> Sorts Σ -> Sorts Σ' ;
```

```
  on_funcs : tagged_morphism (lift_ty on_sorts) (Funcs Σ) (Funcs Σ') ;
  on_preds : tagged_morphism (map on_sorts) (Preds Σ) (Preds Σ')
}.
```

Nothing terribly surprising is happening here. With signature morphisms defined, we can define reduct algebras (DEFINITION 2.4.10), which are a bit more complicated.

```
Definition ReductAlgebra (A' : Algebra Σ') : Algebra Σ.                3.6.17
  refine {|
    interp_sorts s := interp_sorts A' ∘ σ ;
    interp_funcs F θ :=
      rew _ in interp_funcs A' (on_funcs σ F) (rew _ in reindex σ θ) ;
    interp_preds P θ :=
      interp_preds A' (on_preds σ P) (rew _ in reindex σ θ) ;
  |}; rewrite tagged_morphism_commutes; auto.
Defined.
```

Recall the reindexing function from SECTION 3.4. The reason we need it is because the type of θ is $\mathsf{HList}(A' \circ \sigma, \mathsf{ar}\ F)$, which is a Σ-environment, but we need it to be $\mathsf{HList}(A', \mathsf{ar}(\sigma(F)))$, which is a Σ'-environment. This involves two stages—reindexing converts $\mathsf{HList}(A' \circ \sigma, \mathsf{ar}\ F)$ to $\mathsf{HList}(A', \mathsf{map}\ \sigma\ (\mathsf{ar}\ F))$, and the rewrite uses tagged morphism commutativity to convert that to $\mathsf{HList}(A', \mathsf{ar}(\sigma(F)))$.

Term translations are not completely trivial but pose no problems. To define it we will need to also define a reindexing function for the member type we defined in SECTION 3.4.

```
Fixpoint reindex_member                                                3.6.18
    {J K} {j js}
    (f : J -> K) (m : member j js)
    : member (f j) (map f js) :=
  match m with
  | HZ => HZ
  | HS m' => HS (reindex_member f m')
  end.

Equations? on_terms
    {Σ Σ' : Signature} {Γ : Ctx Σ} (σ : SignatureMorphism Σ Σ')
    [s : Sorts Σ] (t : Term Σ Γ s) : Term Σ' (map σ Γ) (σ s) := {
```

```
  on_terms σ (var i) := var (reindex_member σ i) ;
  on_terms σ (term F args) :=
    rew _ in term (on_funcs σ F)
      (rew [HList (Term Σ' (map σ Γ))] _ in map_on_terms σ args)
}
where map_on_terms {Σ Σ' : Signature} {Γ : Ctx Σ} {w : Arity Σ}
    (σ : SignatureMorphism Σ Σ') (args : HList (Term Σ Γ) w)
    : HList (Term Σ' (map σ Γ)) (map σ w) := {
  map_on_terms σ ⟨⟩ := ⟨⟩ ;
  map_on_terms σ (t ::: ts) := on_terms σ t ::: map_on_terms σ ts
}.
Proof.
  all: rewrite tagged_morphism_commutes; reflexivity.
Defined.
```

Term translations only involve a reindexing of members in the variable case. The term case involves applying σ to the function symbol and then recursively applying it to each subterm, subject to some simple rewrites.

As an example of reduct algebras, let's encode the NAND logic example at the end of SECTION 2.4. First, we require the concepts of 'tagged terms' and 'derived signature morphisms'.

```
Definition TaggedTerm Σ                                          3.6.19
    : Tagged (list (Sorts Σ) * Sorts Σ) := {|
  tagged_data := { x & Term Σ (fst x) (snd x) } ;
  get_tag t := projT1 t ;
|}.
```

This explicitly pairs a term with its arity and result sort such that it becomes tagged. This allows it to stand in place of the function symbols in a signature.

Next, we need derived signature morphisms. These are like regular signature morphisms, but the target for the morphism on function symbols is tagged terms.

```
Record SignatureMorphismᵈ (Σ Σ' : Signature) : Type := {          3.6.20
  on_sortsᵈ :> Sorts Σ -> Sorts Σ' ;
  on_funcsᵈ :
    tagged_morphism (lift_ty on_sortsᵈ) (Funcs Σ) (TaggedTerm Σ') ;
  on_predsᵈ :
```

```
    tagged_morphism (map on_sortsᵈ) (Preds Σ) (Preds Σ') ;
}.
```

But once we have the concept of derived signatures, derived signature morphisms can be trivially converted into regular signature morphisms with a derived signature target.

```
Definition TermSignature (Σ : Signature) : Signature := {|          3.6.21
  Sorts := Sorts Σ ;
  Funcs := TaggedTerm Σ ;
  Preds := Preds Σ ;
|}.

Local Notation TS := TermSignature.

Definition collapse_derived {Σ Σ' : Signature}
    (σᵈ : SignatureMorphismᵈ Σ Σ')
    : SignatureMorphism Σ (TS Σ') := {|
  on_sorts := on_sortsᵈ σᵈ : Sorts Σ -> Sorts (TS Σ') ;
  on_funcs := @on_funcsᵈ Σ Σ' σᵈ ;
  on_preds := @on_predsᵈ Σ Σ' σᵈ ;
|}.
```

Finally we can encode the term translation indicated at the end of SECTION 2.4. The most important component converts boolean function symbols to NAND function symbols.

```
Local Notation nt := (term (Σ := nand_sig)).                        3.6.22

Equations b2n_funcs (F : Funcs bool_sig)
                : Term nand_sig (map idmap (ar F)) (res F) :=
  b2n_funcs NOT := nt NAND ⟨ var x₁ ; var x₁ ⟩ ;
  b2n_funcs IMP := nt NAND ⟨ var x₁ ; nt NAND ⟨ var x₂ ; var x₂ ⟩ ⟩ ;
  b2n_funcs OR  := nt NAND ⟨ nt NAND ⟨ var x₁ ; var x₁ ⟩
                           ; nt NAND ⟨ var x₂ ; var x₂ ⟩ ⟩ ;
  b2n_funcs AND := nt NAND ⟨ nt NAND ⟨ var x₁ ; var x₂ ⟩
                           ; nt NAND ⟨ var x₁ ; var x₂ ⟩ ⟩ ;
  b2n_funcs bTRUE  := nt nTRUE  ⟨⟩ ;
  b2n_funcs bFALSE := nt nFALSE ⟨⟩ .
```

This induces a derived signature morphism bool_to_nand^{d}. Now note that any algebra automatically induces an algebra for its derived signature in the following sense.

```
Definition alg_to_tsalg {Σ}                                          3.6.23
    (M : Algebra Σ) : Algebra (TermSignature Σ) := {|
  interp_sorts := interp_sorts M : Sorts (TermSignature Σ) -> Type ;
  interp_funcs F args := eval_term M (projT2 F) args ;
  interp_preds P args := interp_preds M P args ;
|}.
```

This means that the boolean algebra bool_alg is simple to define, assuming that we have defined the NAND algebra nand_alg.

```
Definition bool_alg : Algebra bool_sig :=                            3.6.24
  ReductAlgebra (collapse_derived bool_to_nandᵈ)
                (alg_to_tsalg nand_alg).
```

The full details can be seen at Examples.v#L157.

3.7 *Proofs for First-Order Logic*

We now have everything we need to start proving that *FOPEQ* is an institution. There are broadly four things to prove: that signatures and signature morphisms form a category, that the sentence and model constructions do in fact induce functors into their respective categories Set and Cat—and finally the satisfaction condition. Almost all proofs revolve around signature morphisms.

3.7.1 *The category of signatures*

Let's begin by proving that signatures and signature morphisms form a category. (The obligations are listed in DEFINITION 2.1.1.) This will allow us to introduce some common proof techniques in a simple context. We won't show all the proofs here, just those that best represent the most important ideas.

Two important proofs are the left and right identities for composition for signature morphisms: $f \circ \mathsf{id} = f$ and $\mathsf{id} \circ f = f$. We will apply the same technique as we did near the end of SECTION 3.3, providing an explicit equality lemma that lists sufficient conditions for equality of signature morphisms.

```
Lemma eq_signature_morphism (σ σ' : SignatureMorphism Σ Σ')        3.7.1
  (p : on_sorts σ = on_sorts σ')
  (q : rew [λ f, tagged_morphism (lift_ty f) (Funcs Σ) (Funcs Σ')] p
       in (@on_funcs _ _ σ) = @on_funcs _ _ σ')
  (r : rew [λ f, tagged_morphism (map f) (Preds Σ) (Preds Σ')] p
       in (@on_preds _ _ σ) = @on_preds _ _ σ')
  : σ = σ'.
Proof.
  destruct σ, σ'; cbn in *.
  now destruct p, q, r.
Qed.
```

We can see by the lemma that in order to prove $\sigma = \sigma'$, we need three proofs—the first is that the two signature morphisms are equal on sorts, the second is that they are equal on function symbols given that they are equal on sorts, and the third is that they are equal on predicates given that they are equal on sorts.

In this particular case, this is exactly what happens by default if one simply unfolds definitions and applies Coq's built-in f_equal tactic. We opt for a more explicit approach for consistency and because it's easier to generalise and modify if necessary. Furthermore, since no unfolding is required, it should be easier incorporate into an automated decision procedure. (To see more interesting examples of equality lemmas, either return to SECTION 3.3 or skip ahead to SECTION 4.4.) Here is how we use this lemma to prove the left identity law for first-order signatures.

```
Theorem id_left_FOSig (σ : SignatureMorphism Σ Σ') :              3.7.2
  comp_FOSig (id_FOSig Σ') σ = σ.
Proof.
  unshelve eapply eq_signature_morphism; auto;
  now apply tagged_morphism_eq.
Qed.
```

The lemma generates three cases; the first case is proved by reflexivity, and since the two tagged morphisms involved are equal by computation, the remaining two cases are discharged immediately after applying the relevant lemma.

Associativity of composition is proved by precisely the same basic decision procedure.

```
Lemma comp_assoc_FOSig                                              3.7.3
    (A B C D : Signature)
    (h : SignatureMorphism C D)
    (g : SignatureMorphism B C)
    (f : SignatureMorphism A B) :
  comp_FOSig h (comp_FOSig g f) = comp_FOSig (comp_FOSig h g) f.
Proof.
  unshelve eapply eq_signature_morphism; auto;
  now apply tagged_morphism_eq.
Qed.
```

The remaining proofs are either trivial or similar to the previous two. All that is left is to pack the definition up into a category.

```
Definition FOSig : Category.                                        3.7.4
  refine {|
    obj := Signature ;
    hom := SignatureMorphism ;
    homset := Morphism_equality ;
    id := id_FOSig ;
    compose := @comp_FOSig ;
    compose_respects := _ ;
    id_left := @id_left_FOSig ;
    id_right := @id_right_FOSig ;
    comp_assoc := comp_assoc_FOSig ;
    comp_assoc_sym := comp_assoc_sym_FOSig ;
  |}; repeat intro; congruence.
Defined.
```

The leftover proofs discharged at the end by congruence are from the compose_respects setoid requirement. Normally one can choose the

sort of equivalence that morphisms in a category will use—we choose standard type-theoretic equality, effectively ignoring setoids, and we will continue to do so for the remainder of the thesis. This is also the last time we will show the packing of proofs and definitions into a single object, since it takes up far too much space and adds very little to the presentation.

3.7.2 *The sentence functor*

The first-order sentence functor is simple to define. The component on objects is just the sentence construction itself, given in LISTING 3.6.9. The component on signature morphisms is as follows.

```
Equations? fmap_FOPEQ {Γ : Ctx A} (σ : A ~{ FOSig }~> B) :          3.7.5
    FOPEQ A Γ ~{ SetCat }~> FOPEQ B (map σ Γ) :=
  fmap_FOPEQ σ (Forall ψ) := Forall (fmap_FOPEQ σ ψ) ;
  fmap_FOPEQ σ (Exists ψ) := Exists (fmap_FOPEQ σ ψ) ;
  fmap_FOPEQ σ (Pred P args) := Pred (on_preds σ P)
                                    (rew _ in map_on_terms σ args) ;
  fmap_FOPEQ σ (And α β) := And (fmap_FOPEQ σ α) (fmap_FOPEQ σ β) ;
  fmap_FOPEQ σ (Or α β)  := Or (fmap_FOPEQ σ α) (fmap_FOPEQ σ β) ;
  [...]
Proof. now rewrite tagged_morphism_commutes. Defined.
```

We just need to prove the functor laws (recall DEFINITION 2.1.2). Here is the statement of the first law.

```
Theorem fmap_id_FOSen : ∀ (φ : FOPEQ Σ Γ),                          3.7.6
  fmap_FOPEQ (id_FOSig Σ) φ = rew [FOPEQ Σ] (map_id Γ)^ in φ.
```

This is the first time we have had to prove an equality of the form $u =$ rew p in v. Some technique is necessary to explicate here. Proceeding by induction on φ gives us a challenging first case. The problem is that the identity proofs appearing in the inductive hypothesis and conclusion prevent us from progressing. Let's explicitly show the proof state in Coq. Everything above the dashed line is part of the context, everything below the dashed line is the goal. Picture a world without identity proofs:

```
IHφ :  fmap_FOPEQ (id_FOSig Σ) φ = φ                                    3.7.7
------------------------------------------------------------------------
(1/1)
Forall (fmap_FOPEQ (id_FOSig Σ) φ) = Forall φ
```

This would follow trivially by rewriting with the induction hypothesis. But instead we must prove:

```
IHφ :  fmap_FOPEQ (id_FOSig Σ) φ =                                      3.7.8
         rew [FOPEQ Σ] (map_id (s :: Γ))^ in φ
------------------------------------------------------------------------
(1/1)
Forall (fmap_FOPEQ (id_FOSig Σ) φ) =
  rew [FOPEQ Σ] (map_id Γ)^ in Forall φ
```

Where to begin? Let's rewrite using the inductive hypothesis first.

```
Forall (rew [FOPEQ Σ] (map_id (s :: Γ))^ in φ) =                        3.7.9
rew [FOPEQ Σ] (map_id Γ)^ in Forall φ
```

But now what?

Recall from SECTION 3.5 that identity proofs have a sort of groupoid structure that can be abused in proofs involving them. Some important consequences were listed in that section. Those facts are encoded in Coq's standard libraries under the following names.

```
Lemma map_subst [A] [P Q : A -> Type]                                   3.7.10
    (f : ∀ x, P x -> Q x)
    [x y] (H : x = y) (t : P x) :
  rew H in f x t = f y (rew H in t).

Lemma rew_map
    [A B] (P : B -> Type) (f : A -> B)
    [x y] (H : x = y) (t : P (f x)) :
  rew [P ∘ f] H in t = rew f_equal f H in t.

Lemma eq_sym_map_distr
    [A B] (f : A -> B) [x y] (p : x = y) :
  f_equal f (p^) = (f_equal f p)^.
```

This gives us some tools for manipulating goals involving identity proofs. For example, after applying map_subst to the goal we get

```
Forall (fmap_FOPEQ (id_FOSig Σ) φ) =                                    3.7.11
Forall (rew [FOPEQ Σ ∘ cons s] (map_id Γ)^ in φ)
```

which can be simplified further by f_equal (the tactic, not the term):

```
fmap_FOPEQ (id_FOSig Σ) φ =                                             3.7.12
rew [FOPEQ Σ ∘ cons s] (map_id Γ)^ in φ
```

Now we can apply rew_map to obtain

```
rew [FOPEQ Σ] (map_id (s :: Γ))^ in φ =                                 3.7.13
rew [FOPEQ Σ] f_equal (cons s) (map_id Γ)^ in φ
```

and finally eq_sym_map_distr to obtain

```
rew [FOPEQ Σ] (map_id (s :: Γ))^ in φ =                                 3.7.14
rew [FOPEQ Σ] (f_equal (cons s) (map_id Γ))^ in φ
```

These terms are equal as long as we can prove

```
map_id (s :: Γ) = f_equal (cons s) (map_id Γ)                           3.7.15
```

There are two ways to do this: the first is just to apply proof irrelevance—simply assert that any two propositions are equal and be done with it. The second way to do it is to re-prove map_id in such a way to make it true (and provable), which is the approach we took. Either way, the proof is now complete.

But that was just one case. Most of the other cases are similar to this one, except for the case on equality of terms. We need an analogous lemma for terms which asserts that the identity signature morphism has no effect.

```
Definition on_terms_id Σ Γ s (t : Term Σ Γ s) :                    3.7.16
  on_terms (id_FOSig Σ) t =
    rew [λ Γ, Term Σ Γ s] (map_id Γ)^ in t.
```

This is not so easy to prove. The problem now is not so much to do with the identity proofs, but with the induction principle Coq generated for the Term type—it's too weak. Inspecting the induction principle Term_ind generated by Coq reveals that it is missing a crucial hypothesis in the term case which asserts the induction predicate $P : \prod_j A_j \to \mathsf{Prop}$ for each element of that term's arguments. This means that we are stuck proving goals with only the following to work with:

```
F :  Funcs Σ                                                       3.7.17
args :  HList (Term Σ Γ) (fst (Funcs Σ F))
______________________________________________________________________
(1/1)
…
```

It is obviously difficult to prove much of anything without an inductive hypothesis. We therefore need to write a custom induction principle. We will first need to define HForall, which asserts a predicate P for each element of a h-list.

```
Context [J : Type].                                                3.7.18
Context [A B C : J -> Type].

Fixpoint HForall
    (P : ∀ j, A j -> Prop)
    {js : list J} (v : HList A js) : Prop :=
  match v with
  | ⟨⟩ => True
  | x ::: xs => P _ x ∧ HForall P xs
  end.
```

With that defined, we can form a new induction principle for terms, called term_ind′.

```
Context (Σ : Signature) (Γ : Ctx Σ).                               3.7.19
Context (P : ∀ s, Term Σ Γ s -> Prop).
```

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

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

The full proof of this fact is still somewhat delicate, even with this new and improved induction principle. Here it is in full:

```
Definition on_terms_id Σ Γ s (t : Term Σ Γ s) :                    3.7.20
  on_terms (id_FOSig Σ) t = rew [λ Γ, Term Σ Γ s] (map_id Γ)^ in t.
Proof.
  induction t using term_ind'.
  - simp on_terms. rewrite (map_subst (λ _, var)).
    now rewrite reindex_member_id.
  - simp on_terms.
    simpl on_funcs.
    rewrite map_on_terms_hmap; cbn.
    rewrite reindex_id; cbn. rewrite rew_compose.
    revert H; simplify_eqs; intros H.
    setoid_rewrite (proj1 (map_ext_HForall _ _ _) H).
    case eqH; cbn.
    now rewrite hmap_id.
Qed.
```

There are many facts used here which we have not mentioned before. The facts about h-lists and about reindexing member can be found in Basics/HList.v, FOL/Algebra.v, and Institutions/InsFOPEQ.v. The lemma rew_compose expresses the fact that $q_*(p_*(t)) = (p \cdot q)_*(t)$.

3.7.3 *The model functor*

Mercifully, the model reduct functor is much simpler to construct and the proofs are very straightforward. Recall the definition of *algebra homomorphism* (DEFINITION 2.4.5). This is represented as follows.

```
Class AlgHom (h : ∀ s, A s -> B s) :=                                    3.7.21
  alg_hom_commutes : ∀ (F : Funcs Σ) (args : HList A (ar F)),
    h (res F) (interp_funcs A F args) =
      interp_funcs B F (hmap h args) .
```

We did not do so in SECTION 2.4, but we can define identities and composition for homomorphisms easily.

```
Definition AlgebraHom [Σ] (A B : Algebra Σ) :=                           3.7.22
  { h | AlgHom A B h }.

Lemma eq_alghom [Σ] (A B : Algebra Σ) (f g : AlgebraHom A B)
  (p : `f = `g) : f = g.
Proof.
  destruct f, g; cbn in p.
  now apply subset_eq_compat.
Qed.

Definition id_alghom [Σ] (A : Algebra Σ) : AlgebraHom A A.
  exists (λ _, idmap).
  repeat intro; rewrite hmap_id; auto.
Defined.

Definition comp_alghom [Σ] (A B C : Algebra Σ)
  (f : AlgebraHom B C) (g : AlgebraHom A B) : AlgebraHom A C.
Proof.
  exists (λ s x, `f s (`g s x)).
  repeat intro; rewrite hmap_hmap, (proj2_sig g), (proj2_sig f); auto.
Defined.
```

This is sufficient to define the category of algebras. The proofs are not worth repeating here—they can be found at Institutions/InsFOPEQ.v. The model functor itself only really requires two lemmas (which we do not prove here).

```
Lemma reduct_id (Σ : Signature) (A : Algebra Σ) :                3.7.23
  ReductAlgebra (id_FOSig Σ) A = A.

Lemma reduct_comp {A B C : FOSig}
    (g : B ~> C) (f : A ~> B) (M : Algebra C) :
  ReductAlgebra (comp_FOSig g f) M =
    ReductAlgebra f (ReductAlgebra g M).
```

3.7.4 *The satisfaction condition*

The satisfaction condition for first-order logic can be distilled into a single lemma.

3.7.1 *Lemma.* Let $\sigma : \Sigma_1 \to \Sigma_2$ be a signature morphism, let t_1 be a Σ_1-term with Σ_1-context Γ_1, and let A_2 be a Σ_2-algebra. Let $\theta : \mathsf{HList}(A_2|_\sigma, \Gamma_1)$ be a valuation of the variables in Γ_1. Then

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

Since θ is a h-list, the action of σ on θ is just a reindexing; hence we obtain $\sigma(\theta) : \mathsf{HList}(A_2, \mathsf{map}\,\sigma\,\Gamma_1)$—now a Σ_2-environment. The specific requirement generated by the proof of satisfaction for one of the atomic sentences, $t_1 = t_2$, is

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

Compare this with the definition of satisfaction in DEFINITION 2.3.1. Notice that LEMMA 3.7.1 is a strict strengthening of $(*)$, since it shows in fact that the terms under analysis are equal when interpreted. In fact LEMMA 3.7.1 is a strict strengthening of the satisfaction condition for first-order logic when restricted to equality of terms. We can make this fact much more obvious by understanding it as a technically complete presentation of a fact which we should certainly hope is true:

$$(A \circ \sigma)(t) = A(\sigma(t))$$

So we do not just have the satisfaction condition—a mere logical equivalence—but something much stronger: *syntactical equality* of sentences.

The statement of the lemma in Coq is as follows.

```
Lemma helper_eval_term_reduct                                    3.7.24
    (Σ Σ' : FOSig) (A' : Algebra Σ')
    (σ : Σ ~> Σ') {Γ s} (t : Term Σ Γ s) env :
  eval_term (ReductAlgebra σ A') t env =
    eval_term A' (on_terms σ t) (reindex σ env).
```

The proof can be found at Institutions/InsFOPEQ.v#L340. It is a little finicky, but otherwise can be proved directly from what we have done thus far. Finally, the proof of the satisfaction condition itself, which is *very* long, can be found at Institutions/InsFOPEQ.v#L482. I hope you will agree that it is mostly uncomplicated besides the facts presented here.

With *FOPEQ* completely constructed, we have done most of the hard work. Many institutions—and more than just those considered here—either build directly on *FOPEQ* or otherwise use the basic constructions and proofs comprising it. We will se the fruits of this labour in the next chapter.

FOUR

An Institution for Event-B

In this chapter, we will describe and formalise an institution for Event-B in Coq—or rather, a simplified version, a 'core' institution consisting only of first-order logic and variable-update statements. We prove its satisfaction condition and identify the most important facts involved. The work in this chapter first appeared in a much abridged form in [RM22].

4.1 *A Short Description of the Event-B Language*

For more details, see Thai Son Hoang's summary of the Event-B language in [RT13, Appendix A] and Jean-Raymond Abrial's book on Event-B [Abr10]. Event-B is a formal language for describing and modelling discrete state-transition systems. Event-B models consist of *contexts* and *machines*. Contexts contain only first-order data—carrier sets, constants, axioms, theorems. Machines consist of *state variables* and a list of *events*. Each event describes an update to the state and may only fire under certain conditions on the state, given by the event's *guard*. Events with true guards in some state are said to be *enabled* in that state. There are two special events: an *initialisation* event, which sets the initial state of the machine, and the *skip* event, which does nothing. A machine may have a list of *invariants*, which are state predicates that must be true at initialisation and remain true after any event is executed. The 'operation' of a machine goes like this—

1. Initialise the state of the machine.

2. Nondeterministically choose an enabled event and execute it.
3. Repeat step 2 indefinitely.

Event-B models are not really executed, however, since they are only models—they just describe the behaviour of a machine at varying levels of abstraction.

Each Event-B machine generates a list of proof obligations which must be satisfied. In the following, let v and v' be the pre- and post-state variables, let x be the input variables for an event, let K be the initialisation predicate, let I be the invariant predicate, let G be the guard, and let BA be the before–after predicate representing the action of the event. The most important examples of Event-B proof obligations are—

- *Invariant establishment*—the initialisation event should set the state variables to values which make the invariants true. This is written

$$K(v') \vdash I(v') \tag{IE}$$

- *Invariant preservation*—the invariants should remain true after each event is executed. This is written

$$I(v), G(x, v), BA(x, v, v') \vdash I(v') \tag{IP}$$

But the main appeal of Event-B is its support for *syntactical machine refinement*. The basic notion of 'refinement' originated, as far as I can tell, in Dijkstra's 1968 paper on constructive approaches to program correctness [Dij68]. The idea as it appears in modern writing is to begin with a specification for a program and introduce details step-by-step, eventually arriving at a concrete program that can be executed and that adheres to the original specification. Event-B provides explicit support for this process—it generates proof obligations that ensure concrete machines are in fact valid refinements of the abstract machines. Such obligations are generated by *gluing invariants*.

* * *

Prior to Marie Farrell's work on an institution for Event-B [Far17], there was no formal semantics for Event-B. The proof obligations outlined above can be regarded as a 'semantics' for Event-B, in the same way that first-order proof rules can be regarded as providing a semantics for first-order logic. But there is a much more obvious semantics one can outline for an Event-B machine, as we have done at the beginning of this section. Event-B models don't 'do' anything but they do specify a range of possible concrete machines, and therefore can constrain *behaviours* or *execution traces*. (We will return to this concept in CHAPTER 5.) A semantics for Event-B should reflect the more basic literal meaning of the syntax.

An important result that Farrell proved was the *amalgamation property*—meaning that Event-B can in principle support the modularisation constructs it currently lacks. It means, roughly, that models for combined signatures can themselves be put together from models for the individual signatures. For example, if A and B are signatures and $C = A \cap B$ is a signature consisting of the symbols shared by both, then models for $A \cup B$ are given by combining models for A and B individually, as long as the models agree on the common symbols C. (This will be defined precisely in DEFINITION 6.4.2.)

The institution we will discuss in this chapter shares most of the important features of Farrell's EVT, but removes features that were unimportant for proving the satisfaction condition. We do this because it simplifies the presentation, retains most of the work involved, and because it is a useful generalisation of EVT. This institution is not just useful for modelling Event-B, but potentially *any* logical system which is based on the notion of variable update, like TLA$^+$ [Lam99].

We will therefore not encode Farrell's EVT directly for fear of needlessly distending this chapter. We will only discuss Farrell's construction to compare it to the constructions we detail here. Besides, we have encoded EVT already—see [Rey21] for that.

4.1.1 *Event-B machine example: traffic lights*

We'll quickly define the standard traffic lights example to give a flavour of the Event-B language. It consists of two state variables cars_go and peds_go, three invariants, and five events including one initial event. The notation should be self-explanatory. (But note that this is not quite how Event-B appears in the Rodin tool.)

```
machine traffic_lights                                          4.1.1
  variables cars_go, peds_go

  invariants
    @inv1: cars_go ∈ BOOL
    @inv2: peds_go ∈ BOOL
    @inv3: ¬ (cars_go = true ∧ peds_go = true)

  events
    initialisation
    begin @act1: cars_go := false
          @act2: peds_go := false
    end

    set_cars_go
    when @grd1: peds_go = false
    then @act1: cars_go := true
    end

    set_peds_go
    when @grd1: cars_go = false
    then @act1: peds_go := true
    end

    set_cars_stop
    then @act1: cars_go := false
    end

    set_peds_stop
    then @act1: cars_go := false
    end
```

The two state variables are declared and constrained to be booleans in the first and second invariants. The primary invariant is inv3, which says that it must not be the case that cars_go and peds_go are true at once.

This is not a remotely plausible model of any real set of traffic lights, but it does capture the key abstract requirement that a set of traffic lights must satisfy: cars and pedestrians must not go at once. Details can be introduced through refinement while retaining this basic requirement through gluing invariants. First, the new machine will need a context providing the colours red and green for the lights.

```
context ctx                                                    4.1.2
  sets colours
  constants red, green
  axioms
    @axm1: partition(colours, {red}, {green})
end
```

Now, here is a possible refinement of the traffic_lights machine.

```
machine traffic_lights_r1                                      4.1.3
    refines traffic_lights
    sees ctx

  invariants
    @inv1: peds_colour ∈ {red, green}
    @inv2: cars_colour ∈ {red, green}
    @inv3: button_pushed ∈ BOOL
    @inv4: peds_go = true ⟺ peds_colour = green
    @inv5: cars_go = true ⟺ cars_colour = green

  events
    initialisation
    begin @act1: cars_colour := red
          @act2: peds_colour := red
    end

    set_peds_green refines set_peds_go
    when @grd1: cars_colour = red
         @grd2: button_pushed = true
```

```
then @act1: peds_colour := green
     @act2: button_pushed := false
end

set_peds_red refines set_peds_stop
begin @act1: peds_colour := red
end

set_cars_green refines set_cars_go
when @grd1: peds_colour = red
then @act1: cars_colour := green
end

set_cars_red refines set_cars_stop
begin @act1: cars_colour := red
end

press_button
begin @act1: button_pushed := true
end
```

The invariants inv4 and inv5 are the gluing invariants—they ensure that cars and pedestrians go if and only if their respective lights are green. This, combined with the original invariant inv3 in the traffic_lights machine, ensures that the refinement preserves the old invariant—if we can prove it, of course.

4.2 *A Simplified Institution for Event-B*

The institution we will describe in this section is similar to Farrell's, but without event names. We focus only on the description of variable updates via assignments of the form $x := E$. These assignments will be translated as equations $x' = E$, where E is an expression involving only 'unprimed' variables. The unprimed variables represent the state of the machine 'before' the assignment, and the primed variables represent the state of the machine 'after' the assignment. There will be two types of assignments: *initialisation* and *event* assignments. Initialisation assignments only mention primed variables and serve, unsurprisingly, to initialise the

state. Event assignments mention both primed and unprimed variables and describe updates to the state.

Stripped in this way, the sentences in this institution represent two-state predicates. It is akin to Leslie Lamport's temporal logic of actions, but without the temporal operator $\Box$, meaning 'forever' [Lam99; Pnu77], and modelled not against a behaviour—i.e. an execution trace—but against a single pair of states. It represents a minimal core in which we may talk about assignment, or 'actions' in Lamport's terminology. Hence we will call this institution ACT for the purposes of this thesis. This institution went by the name $mEVT$ in our [RM22] but was changed to emphasise the general character of ACT. Similar institutions to ACT appear in [Kna+15] and [Ros+21, §3.1].

Before we start defining ACT's components, we need to introduce a few standard first-order constructions. One such standard construction in institution theory is the *signature extension* $\Sigma \to \Sigma + X$. We can add a set of variables X to a signature Σ by adding them directly into the signature as constant function symbols.

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

We can extend algebras in much the same way. An algebra for a signature of the form $\Sigma + X$ consists exactly of a Σ-algebra and a valuation $X \to A$.

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

Let's now define signatures and signature morphisms for ACT.

4.2.3 *Definition.* An *ACT-signature* $\hat{\Sigma}$ is a 3-tuple $\langle \Sigma, X, X' \rangle$ where Σ is a first-order signature and X and X' are $\mathsf{Sorts}(\Sigma)$-indexed sets containing respectively unprimed and primed variables, such that the function $(-)' : X \to X'$ mapping variables to their primed counterparts is an equivalence. From now on, in all cases where not otherwise specified, any *ACT*-signature $\hat{\Sigma}_i$ is given by $\langle \Sigma_i, X_i, X_i' \rangle$.

Notice that the variables are not pooled into one set, but separated. Despite appearances to the contrary, there is no difference between this presentation of variables and Farrell's. In Farrell's *EVT*, where there is only one set X of variables, one can always retrieve the primed version of a variable x. That implies that the set contains variables $x, y, z, \ldots$ and their primed counterparts $x', y', z' \ldots$ *and* that there is a partial function $(-)' : X \rightharpoonup X$ defined only on unprimed variables that maps each unprimed variable to its primed counterpart. Here, we simply make these facts explicit.

General functions on sorted values often have an underlying signature morphism. In those situations the arrow will be written $A \xrightarrow{\sigma} B$. The most common example is a morphism of variables, as we will see next.

4.2.4 *Definition.* An *ACT-signature morphism* $\hat{\sigma} : \hat{\Sigma}_1 \to \hat{\Sigma}_2$ consists of a first-order signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ and two variable morphisms $\mathsf{on\_vars} : X_1 \xrightarrow{\sigma} X_2$ and $\mathsf{on\_vars}' : X_1' \xrightarrow{\sigma} X_2'$ such that the following diagram commutes.

$$\begin{array}{ccc} X_1 & \xrightarrow{\mathsf{on\_vars}} & X_2 \\ {\scriptstyle (-)'}\uparrow\downarrow & & \uparrow\downarrow{\scriptstyle (-)'} \\ X_1' & \xrightarrow[\mathsf{on\_vars}']{} & X_2' \end{array}$$

This commutativity condition can also be characterised by the equation $\mathsf{on\_vars}'(x') = (\mathsf{on\_vars}(x))'$.

4.2.5 *Definition.* Let $\hat{\Sigma}$ be an *ACT* signature and let A be a Σ-algebra. A $\hat{\Sigma}$-*state* is a valuation of variables $\theta : \prod_s X_s \to A_s$. Normally the sort

is left implicit and we just write $\theta : X \to A$. We will sometimes call Σ-states 'valuations' or 'environments'.

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

The sentences for ACT are very simple and correspond to the two cases noted in the previous section. Note that $\mathsf{FOSen}(\Sigma)$ denotes the set of all first-order Σ-sentences.

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

To illustrate what we have so far, consider the following example. A model for a $(\Sigma + X + X')$-sentence consists of a Σ-algebra A and two states $\theta : X \to A$ and $\theta' : X' \to A$. 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 $s' = \mathsf{push}(x, s)$ consists of an algebra for a stack of characters, a pre-state $\{x \mapsto \mathrm{e}, s \mapsto [\mathrm{v}, \mathrm{t}]\}$, and a post-state $\{s' \mapsto [\mathrm{e}, \mathrm{v}, \mathrm{t}]\}$. Here, x' can consistently be assigned anything. If we wish to avoid this, we can assume that sentences ψ which don't mention a primed variable x' are really shorthand for $\psi \wedge (x' = x)$.

Now we can define the semantic entailment relation for ACT.

4.2.8 *Definition.* Let $\hat{\Sigma}$ be an ACT-signature, let $M = \langle A, \theta, \theta' \rangle$ a $\hat{\Sigma}$-model, and let ψ an ACT-sentence. We define $M \models \psi$ by cases—

- $M \models \mathsf{Init}(\phi)$ if $A^{\theta'} \models \phi$; and
- $M \models \mathsf{Event}(\phi)$ if $A^{\theta + \theta'} \models \phi$

This is as simple as it can possibly get—the only simpler representation might be to collapse the initial and event distinction entirely, which doesn't seem necessary. All that remains is to encode everything in Coq and prove the satisfaction condition.

4.3 *Formalising ACT in Coq*

Since *ACT* builds directly on *FOPEQ*, we have far less work to do than for *FOPEQ*. We rely on a couple of major first-order constructions. First, we will define *ACT* state variables, which are represented as tagged data with decidable equality.

```
Record Vars Σ := {                                                    4.3.1
  tvars :> Tagged (Sorts Σ) ;
  vars_dec : EqDec (tagged_data tvars) eq ;
}.
```

Variable morphisms are just a specialisation of the tagged morphisms of SECTION 3.3.

```
Definition var_morphism (σ : Sorts A -> Sorts B)                      4.3.2
  (X : Vars A) (Y : Vars B) := tagged_morphism σ X Y.
```

Then we define the following class.

```
Class Primed [Σ] (X Y : Vars Σ) := {                                  4.3.3
  prime : var_morphism id{FOSig} X Y ;
  unprime : var_morphism id{FOSig} Y X ;
  p_unp : ∀ y : Y, prime (unprime y) = y ;
  unp_p : ∀ x : X, unprime (prime x) = x ;
}.
```

This encodes the information that X and Y are equivalent as types. *ACT*-signatures (DEFINITION 4.2.3) are then represented exactly as they are given mathematically.

```
Record EvtSignature := {                                              4.3.4
  base :> Signature ;
  vars  : Vars base ;
  vars' : Vars base ;
  prime_rel : Primed vars vars' ;
}.
```

For *ACT* signature morphisms (DEFINITION 4.2.4), we can define on_vars′ in terms of on_vars to simplify matters.

```
Record EvtSigMorphism (Σ Σ' : EvtSignature) : Type := {          4.3.5
  on_base :> SignatureMorphism Σ Σ' ;
  on_vars : var_morphism on_base (vars Σ) (vars Σ')
}.

Definition on_vars' (σ : EvtSigMorphism Σ Σ') :
  var_morphism σ (vars' Σ) (vars' Σ').
Proof.
  refine (exist _ (λ x, prime (on_vars σ (unprime x))) _); intros.
  refine (eq_trans _ _). { apply var_morphism_commutes. }
  refine (eq_trans _ _). { apply var_morphism_commutes. }
  apply f_equal, var_morphism_commutes.
Defined.
```

The proof for on_vars′ might look a little strange, but proving it exactly like this is useful in situations where the structure of the proof term matters—recall the discussion in SECTION 3.5.

Signature extensions (DEFINITION 4.2.1) are easy to define—recall that we wanted to treat the variables like constant function symbols and take their sum.

```
Definition SigExtension (Σ : FOSig) (X : Vars Σ) : FOSig := {|   4.3.6
  Sorts := Sorts Σ ;
  Funcs := tagged_sum (Funcs Σ) (as_constant_funcs X) ;
  Preds := Preds Σ ;
|}.
```

Valuations of variables (DEFINITION 4.2.5) are internally referred to as 'environments' and are defined as follows.

```
Definition Env [Σ] (X : Vars Σ) (A : Sorts Σ -> Type) :=         4.3.7
  ∀ (x : X), A (get_tag x).
```

We noted in DEFINITION 4.2.5 that when we write $X \to A$ we mean $\prod_s X_s \to A_s$, but with our representation of indexed types (recall SECTION 3.3) it is more precisely

$$\prod_{x:X} A_{\mathsf{tag}(x)}$$

We will still write $X \to A$ for clarity.

Environments, like algebras, are retracted along signature morphisms, and the retraction is just precomposition. Let $\theta : Y \to M$ be a valuation and $f : X \xrightarrow{\sigma} Y$ be a variable morphism. The precomposition $\theta \circ f$ should have the type $X \to M \circ \sigma$, but instead has the type

$$\prod_{x:X} M(\mathsf{tag}(f(x)))$$

This type is not the right shape to be an environment. Recall however that $\mathsf{tag}(f(x)) = \sigma(\mathsf{tag}\ x)$ by the coherence condition for the tagged morphism f. Hence the above is propositionally equal to

$$\prod_{x:X} M(\sigma(\mathsf{tag}\ x)) \triangleq \prod_{x:X} (M \circ \sigma)(\mathsf{tag}\ x))$$

which is the type we expect for an environment $X \to M \circ \sigma$. The full definition is hence—

```
Definition retract_env                                                  4.3.8
    [A B : FOSig] [X : Vars A] [Y : Vars B]
    [M : Sorts B -> Type]
    (σ : A ~> B) (f : var_morphism σ X Y) (θ : Env Y M)
    : Env X (M ∘ σ) :=
  λ x, rew (proj2_sig f x) in θ (f x).
```

The most important part of an algebra expansion (DEFINITION 4.2.2) is given by the following function—no other part of the algebra is changed. The environment θ interprets the variables and the algebra interprets everything else.

```
Equations alg_exp_funcs {Σ : FOSig} {X : Vars Σ}                        4.3.9
    (A : Algebra Σ) (θ : Env X A) (F : Funcs (SigExtension Σ X) )
    : HList (interp_sorts A) (ar F) -> interp_sorts A (res F) :=
  alg_exp_funcs _ _ (inr C) := λ _, θ C ;
  alg_exp_funcs _ _ (inl F) := interp_funcs A F .
```

ACT-models (DEFINITION 4.2.6) and *ACT*-sentences (DEFINITION 4.2.7) also offer no surprises.

```
Record EvtModel Σ := {                                          4.3.10
  base_alg :> Algebra Σ ;
  env  : Env (vars Σ) base_alg ;
  env' : Env (vars' Σ) base_alg ;
}.
```

```
Inductive EVT Σ : Type :=                                       4.3.11
| Init  : FOSen (SigExtension Σ (vars' Σ)) -> EVT Σ
| Event : FOSen (SigExtension Σ (vars Σ ⊕ vars' Σ)) -> EVT Σ.
```

Finally, the semantic entailment relation for ACT (DEFINITION 4.2.8) defers directly to entailment for $FOPEQ$.

```
Definition interp_evt (M : EvtModel Σ) (φ : EVT Σ) : Prop :=    4.3.12
  match φ with
  | Init ψ =>
      AlgExpansion (base_alg M) (env' M) ⊨ ψ
  | Event ψ =>
      AlgExpansion (base_alg M) (join_envs (env M) (env' M)) ⊨ ψ
  end.
```

Here, join_envs stitches two valuations $\theta : X \to A$ and $\theta' : X' \to A$ (with the same target) into a valuation $(\theta + \theta') : X + X' \to A$. The definition is simple.

```
Definition join_envs                                            4.3.13
    {Σ : FOSig} {X X' : Vars Σ} {M : Algebra Σ}
    (θ : Env X M) (θ' : Env X' M) : Env (X ⊕ X') M :=
  λ x, match x with
  | inl x => θ x
  | inr x => θ' x
  end.
```

With the basic concepts defined we can move on to the proofs.

4.4 *Proofs for ACT*

Recall in SECTION 3.3 that we introduced explicit equality lemmas. Their merits will become much more apparent in this section.

To prove, for example, that two ACT signature morphisms are equal, we need to prove that they are equal componentwise—the first-order signature morphisms must agree everywhere, as must the two variable morphisms. The usual strategy is to unfold definitions, split products, and apply the f_equal tactic. But this produces a strange set of subgoals that happen to be difficult to prove. What's the problem?

Let's write out the equality lemma explicitly, as we have done before.

```
Lemma eq_evtsigmorphism                                          4.4.1
  (Σ Σ' : EvtSignature) (σ σ' : EvtSigMorphism Σ Σ')
  (p : on_base σ = on_base σ')
  (q : rew [λ f, var_morphism f (vars Σ) (vars Σ')] p
       in on_vars σ = on_vars σ')
  : σ = σ'.
```

Notice that in order to state the proof q, we need to know p is true first. The crucial observation is that p is a stronger assumption than we need to state q. The function f in the type family argument to eq_rect is a full first-order signature morphism, but the only component that participates is the operation on sorts. We therefore only need to know that σ and σ' agree on sorts in order to state q. We can add that as an extra assumption p' and state q in terms of p' instead.

```
Lemma eq_evtsigmorphism                                          4.4.2
  (Σ Σ' : EvtSignature) (σ σ' : EvtSigMorphism Σ Σ')
  (p' : on_sorts σ = on_sorts σ')
  (p : on_base σ = on_base σ')
  (q : rew [λ f, var_morphism f (vars Σ) (vars Σ')] p'
       in on_vars σ = on_vars σ')
  : σ = σ'.
```

This lemma is true, and we can prove it easily with UIP.

Using this equality lemma, it's trivial to prove that ACT signatures and signature morphisms form a category. As an example, the left identity law is discharged by the following.

```
unshelve eapply eq_evtsigmorphism; cbn.                          4.4.3
* reflexivity.
```

```
* apply id_left_FOSig.
* apply var_morphism_left_id.
```

In fact this is just what we would have *expected* to prove should the complex structure of the types have not hindered progress. The equality lemma requires us to prove three subgoals, and each subgoal has a proof already. Since the proof of p' is refl, eq_rect computes away in the third branch of the proof and we can proceed as usual. All proofs for the ACT signature category are discharged in much the same way.

Next, we will explain how to turn ACT signature morphisms into $FOPEQ$ signature morphisms of extended signatures. (This is what an ACT signature morphism 'really is'—a first-order signature morphism of signatures extended by variables.) We will call this a 'flattening' and define it as follows.

```
Definition flatten_morphism                                    4.4.4
    {Σ₁ Σ₂ : FOSig} (σ : Σ₁ ~> Σ₂)
    {X₁ : Vars Σ₁} {X₂ : Vars Σ₂}
    (v : var_morphism σ X₁ X₂) :
    SigExtension Σ₁ X₁ ~> SigExtension Σ₂ X₂.
Proof.
  refine {|
    on_sorts := on_sorts σ ;
    on_funcs := _ ;
    on_preds := on_preds σ ;
  |}.
  unshelve esplit.
  - intros F. destruct F as [F | x].
    * left. exact (on_funcs σ F).
    * right. exact (v x).
  - intros F. destruct F as [F | x]; cbn.
    * apply tagged_morphism_commutes.
    * unfold lift_ty. f_equal.
      apply tagged_morphism_commutes.
Defined.
```

Another important construction involves stitching together two variable morphisms with the same underlying signature morphism. It's simple to define and the coherence proofs are easy.

```
Definition varmap_sum [A B] [X Y Z W]                                    4.4.5
  (σ : Sorts A -> Sorts B)
  (f : var_morphism σ X Z)
  (g : var_morphism σ Y W)
  : var_morphism σ (X ⊕ Y) (Z ⊕ W).
Proof.
  refine (exist _ (λ x, match x with
  | inl x => inl (f x)
  | inr x => inr (g x)
  end) _); intros.
  destruct x; cbn.
  - rewrite <- (proj2_sig f _); reflexivity.
  - rewrite <- (proj2_sig g _); reflexivity.
Defined.
```

Before we look at the proof of satisfaction for ACT directly, we will note the following important observation.

```
Lemma varmap_sum_join [A B : EvtSig] (σ : A ~> B) M' :                   4.4.6
  retract_env σ (varmap_sum σ (on_vars σ) (on_vars' σ))
    (join_envs (env M') (env' M')) =
  join_envs (env (fmap[EvtMod] σ M')) (env' (fmap[EvtMod] σ M')).
Proof.
  unfold retract_env; funext x;
  destruct x; simplify_eqs; auto.
Qed.
```

This proves that $(\theta + \theta')|_{v+v'} = \theta|_\sigma + \theta'|_\sigma$, which might look a little surprising. In fact, retracting along variable morphisms is the same as taking the reduct of the underlying algebra by the ambient signature morphism, as the following lemmas demonstrate.

```
Lemma varmap_retract [A B : EvtSig] (σ : A ~> B) M' :                    4.4.7
  retract_env σ (on_vars σ) (env M') = env (fmap[EvtMod] σ M').
Proof. reflexivity. Qed.

Lemma varmap_retract' [A B : EvtSig] (σ : A ~> B) M' :
  retract_env σ (on_vars' σ) (env' M') = env' (fmap[EvtMod] σ M').
Proof. reflexivity. Qed.
```

The final and most important lemma is the following. The proof proceeds more-or-less by computation.

```
Lemma expand_retract_eq {A B : FOSig} (σ : A ~> B)                    4.4.8
    {M' : Mod[INS_FOPEQ] B} {X : Vars A} {X' : Vars B}
    {θ' : Env X' M'} {v : var_morphism σ X X'} :
  AlgExpansion (ReductAlgebra σ M') (retract_env σ v θ') =
    ReductAlgebra (flatten_morphism σ v) (AlgExpansion M' θ').
Proof.
  unfold AlgExpansion, ReductAlgebra, flatten_morphism; f_equal.
  funext F θ. destruct F; cbn in *; auto.
  unfold retract_env. simplify_eqs. destruct eqH. now simplify_eqs.
Qed.
```

Loosely it says that expansion and reduction 'commute'. More precisely:

4.4.1 *Lemma.* Let $\sigma : \Sigma_1 \to \Sigma_2$ be a first-order signature morphism, let $f : X_1 \xrightarrow{\sigma} X_2$ be a variable morphism, let A_2 be a Σ_2-algebra, and let $\theta_2 : X_2 \to A_2$ be a valuation of variables. Then

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

This lemma is useful not just for *ACT*, but for other institutions we will define in CHAPTER 5.

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*, LISTING 4.4.6, LEMMA 4.4.1 (which was encoded in LISTING 4.4.8), with f instantiated to different maps in each.

```
Definition INS_EVT : Institution.                                     4.4.9
Proof.
  refine {|
    Sig := EvtSig ;
    Sen := EvtSen ;
    Mod := EvtMod ;
    interp := @interp_evt ;
    sat := _ ;
  |}; intros.
  induction φ; split; intros.
```

```
  (* Init case *)
  - apply expand_retract_iff; auto.
  - apply expand_retract_iff; auto.
  (* Event case *)
  - apply expand_retract_iff in H; unfold interp_evt, "⊨".
    now rewrite varmap_sum_join in H.
  - apply expand_retract_iff. unfold interp_evt, "⊨" in H.
    now rewrite varmap_sum_join.
Qed.
```

That completes the institution ACT.

The hope is that the story of ACT proves something about how straightforward defining institutions based on the components of first-order logic can be. And with both $FOPEQ$ and ACT defined, any institutions based on 'FOL with state' should be simple to describe, even if they do not have any explicit formal relationship with them. That is what we will endeavour to show in the next chapter.

FIVE

Logic Combinations: Event-B and LTL

With first-order logic, state, and basic state-updates encoded, we're now well-placed to iterate on these ideas. In this chapter we'll define a new institution for Event-B called $MacEVT$—which will directly encode the semantics for an Event-B machine—and prove its satisfaction condition in Coq. We will also encode an institution for linear-time temporal logic (LTL) in Coq called LTL, then discuss and evaluate a logic combination of $MacEVT$ and LTL using a duplex construction.

5.1 *Introduction and Motivation*

Event-B models are generally focused on safety properties—'something bad never happens'—rather than liveness properties—'something good eventually happens' [Lam77]. But there is some appetite for more general liveness properties in Event-B—plenty has been written by (for instance) Thai Son Hoang and Jean-Raymond Abrial [HA11] and Steve Schneider *et al.* [Sch+14] about such properties. Indeed, the ProB animator and model checker for Event-B have supported LTL properties for some time now [DLP16]. In this chapter, we will connect Event-B and LTL at the institution level to formally ground the logic combination of the two. Before we discuss the formal combination, let's get a general sense of what the combination might look like. (Skip ahead to SECTION 5.2 for formal definitions of the LTL concepts discussed in this section.)

Linear temporal logic is generally defined over traces and is therefore well-suited to specifying properties of machines. In SECTION 5.2 we will define an institution for LTL, unsurprisingly called *LTL*, which will make this precise. We are specifically interested in modelling the following important kinds of liveness properties for Event-B (adapted from [HA11]):

$$\mathsf{G}([\mathsf{req}] \to \mathsf{F}[\mathsf{ack}]) \tag{5.1}$$

meaning that, whenever the event req is executed, the event ack must eventually be executed—when a request is issued the machine eventually responds.*

In Farrell's thesis [Far17], Event-B machines are *presentations* over *EVT* (see DEFINITION 6.1.2). The sentences of *EVT* represent single events, and their models consist only of a pre- and post-state. But it does not seem to me possible to express the meanings of LTL sentences against such a model. What can $\mathsf{F}\,\phi$—which says that ϕ is eventually true—mean if we can only see a pre-state and a post-state? LTL sentences relate to the entire trace, whereas *EVT* sentences are concerned only with a single step from a pre-state to a post-state. Put another way, LTL sentences are more similar to machines than events. I therefore suspect that a direct logic combination of *EVT* and LTL at the level of institutions should not be possible without some undignified contortion.

One way to link the two is to create a new institution for Event-B in which the sentences are not events, but *machines*. The motivation is simple: LTL sentences and Event-B machines alike constrain whole execution traces, and since the signatures and models are very similar (as we'll see), we should be able to construct a *duplex* combination of the two institutions. This even allows us to reintroduce invariants, so far neglected, in a more natural context. While an invariant ϕ could be modelled as a separate 'machine' that performs no actions and simply monitors the state (so that we need no special syntax to represent it), it

*The globally operator G is often paired with the future or finally operator F in this manner to say that a particular liveness property is satisfied infinitely often—'something good will always eventually happen'.

can be more naturally encoded as the LTL sentence $\mathsf{G}\,\phi$. This suggests that LTL could stand in as a more sophisticated invariant language for Event-B. In place of simple first-order invariants, one can have any LTL sentences whatsoever.

Over the next few sections we will define the institutions *LTL* (SECTION 5.2) and *MacEVT* (SECTION 5.3), then encode them in Coq and prove their satisfaction conditions (SECTIONS 5.4 & 5.5).

5.2 *The Institution for Linear-time Temporal Logic*

We will use a standard presentation of the syntax and semantics for finite-trace LTL, similar to Grigore Roşu's presentation in [Roş18], but we will build LTL over FOL, so that any first-order predicate involving the state variables is a valid LTL sentence. We will encode LTL state variables as constant function symbols in FOL, just like we did for ACT. We will provide the definitions quickly since we will formalise it in Coq in SECTION 5.5.

We present a finite-trace LTL for a couple of reasons, the foremost being for validation of the resulting construction via examples, but also because finite-trace LTL is useful in its own right for model checking and runtime verification. Implementing an infinite-trace LTL would be a simple task given the work to follow by replacing non-empty lists with functions $\mathbb{N} \to (X \to A)$, for example.

While it is possible to describe LTL sentences over an arbitrary institution using a similar trace semantics (see SECTION 6.6), this version is less useful for us in the immediate term—though it might be possible to recover the technique by specifying a general 'institution with state-variables'.

5.2.1 *Definition.* An *LTL signature* consists of a first-order signature Σ and a set of state variables X. These are just like ACT signatures (DEFINITION 4.2.3) but without primed variables.

5.2.2 *Definition.* An *LTL* signature morphism is a pair $\langle \sigma, \mathsf{on\_vars} \rangle$ where $\sigma : \Sigma \to \Sigma'$ is first-order signature morphism and $\mathsf{on\_vars} : X \xrightarrow{\sigma} X'$ is a variable morphism.

5.2.3 *Definition.* *LTL sentences* are given by the following grammar:

$$\phi ::= \mathsf{FOL}(\psi) \mid \mathsf{true} \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{X}\,\phi \mid \phi\ \mathsf{U}\ \phi$$

where X and U are the standard next and until operators, $\neg$, $\wedge$, $\vee$ are the usual propositional connectives (at the LTL level) and ψ is a first-order sentence.

5.2.4 *Definition.* An LTL model consists of a first-order algebra A and a non-empty trace $\pi : \mathsf{NEList}(X \to A)$.

5.2.5 *Definition.* Let A be the underlying first-order algebra. The semantics for *LTL*, $\pi \models \phi$, is defined as follows: First, $\pi \models \mathsf{true}$ is always true. Now let $\pi^j = (e_{i+j}, s_{i+j})_{i \in \mathbb{Z}^+}$ be the truncation of π with the first j elements dropped.

$\pi \models \mathsf{FOL}(\psi)$	if	$A^{\pi_1} \models \alpha$
$\pi \models \neg\alpha$	if	$\pi \models \alpha$ is false
$\pi \models \alpha \vee \beta$	if	$\pi \models \alpha$ or $\pi \models \beta$
$\pi \models \mathsf{X}\,\alpha$	if	$\mathsf{length}(\pi) = 1$ or $\pi^1 \models \alpha$
$\pi \models \alpha\ \mathsf{U}\ \beta$	if	$\pi^i \models \alpha$ for all i, or there is some i such that $\pi^i \models \beta$ and $\pi^j \models \alpha$ for all $j < i$.

The semantics defined here is *weak*. Notice that $\pi \models \mathsf{X}\,\psi$ is vacuously true if there is no next state, and that $\pi \models \alpha\ \mathsf{U}\ \beta$ can be true even if β never becomes true, so long as α holds globally.

Define $\mathsf{false} \triangleq \neg\mathsf{true}$. The until operator subsumes the globally and finally operators, which can be defined as $\mathsf{G}\,\psi \triangleq \psi\ \mathsf{U}\ \mathsf{false}$ and $\mathsf{F}\,\psi \triangleq \neg\,\mathsf{G}\,\neg\psi$. These can also be defined directly rather easily, but it lengthens the proof of satisfaction to include them.

5.3 *The Institution of Machines*

The institution of machines will be called *MacEVT*. As with *LTL*, we'll provide most of the definitions quickly and without much commentary since it will be defined more precisely in Coq in SECTION 5.4.

5.3.1 *Definition.* A *status* is a label given to an event which describes its convergence properties. It is one of $\{\mathsf{ordinary} \lesssim \mathsf{anticipated} \lesssim \mathsf{convergent}\}$.

Statuses determine convergence properties for events. Ordinary events have no convergence properties. Convergent events must eventually collectively cede control—each convergent event must strictly decrease a shared variant expression $V \in \mathbb{N}$. Anticipated events are similar but more relaxed, and must only avoid increasing the variant. The idea is that anticipated events will be shown to converge in a later refinement [RT13, Appendix A].

5.3.2 *Definition.* A *MacEVT signature* is a 2-tuple $\tilde{\Sigma} = \langle \hat{\Sigma}, E \rangle$, where $\hat{\Sigma} = \langle \Sigma, X, X' \rangle$ is an *ACT* signature and E is a status-indexed set of event names, not including the initial event.[†] As usual, as a matter of notation, indices will be inherited by subobjects; for example, E_i is the set of event names of the signature $\tilde{\Sigma}_i$.

5.3.3 *Definition.* A *MacEVT signature morphism* is a pair $\langle \hat{\sigma}, f \rangle$ where $\hat{\sigma} : \hat{\Sigma}_1 \to \hat{\Sigma}_2$ is an *ACT* signature morphism and $f : E_2 \to E_1$ is an event morphism. Notice that the event-name morphism runs against the direction of the *ACT* signature morphism, a fact that will be discussed in more detail in SECTION 5.9.

5.3.4 *Definition.* A *MacEVT sentence* is a pair $\langle \phi_{\mathsf{init}}, \phi \rangle$ which consists of an initialisation sentence $\phi_{\mathsf{init}} : \mathsf{FOSen}(\Sigma + X)$ and a function

$$\phi : E \to \mathsf{FOSen}(\Sigma + X + X')$$

[†]The initial event does not really function as an 'event' in the way other events do, so in this chapter by 'event' we will generally mean 'non-initial event'.

which we will denote $\phi_e = \phi(e)$. The idea is that each event name is assigned exactly one first-order sentence representing the event's guard and action.

5.3.5 *Definition.* A *MacEVT model* is a triple $\langle A, s_0, \pi \rangle$, where A is a first-order algebra, $s_0 : X \to A$ is an initial state, and

$$\pi = (e_i, s_i)_{i \in \mathbb{Z}^+}$$

is an execution trace, where each $s_i : X \to A$ is a state and each $e_i : E$ is a event.

The semantics for machines will defer to entailment for first-order logic at each time step.

5.3.6 *Definition.* Let $q : X' \to X$ be the unpriming function. We'll consider first an auxiliary relation $\models^s \subseteq \mathsf{Mod}(\Sigma) \times \mathsf{Sen}(\Sigma)$ which depends on a given 'current state' $s : X \to A$. If π^j is empty then $\pi^j \models^s \phi$ is true vacuously. If not, then $\pi^j \models^s \phi$ evaluates to

$$A^{s \oplus (s_j \circ q)} \models \phi_{e_j} \quad \text{and} \quad \pi^{j+1} \models^{s_j} \phi$$

Full satisfaction $\models \subseteq \mathsf{Mod}(\Sigma) \times \mathsf{Sen}(\Sigma)$ is therefore defined as

$$A^{s_0} \models_{FOPEQ} \phi_{\mathsf{init}} \quad \text{and} \quad \pi \models^{s_0} \phi$$

A trace models a machine if the initial state is consistent with the machine's initial event, and if the state is stepped consistently with the events in the machine.

The next two sections will formalise the preceding developments in Coq, but if you want some examples for the constructions so far introduced in this chapter, skip ahead to SECTION 5.7.

5.4 *Machines in Coq*

The code for this section can be found at Institutions/Machine.v.

A status is either ordinary, anticipated, or convergent.

```
Inductive Status := ordinary | anticipated | convergent.          5.4.1
```

Machine signatures are *ACT* signatures with events, and the status of an event is just its tag.

```
Record MachineSignature := {                                      5.4.2
  evt_sig :> EvtSig ;
  events : Tagged Status ;
}.

Definition status [Σ] (e : events Σ) : Status :=
  get_tag e.
```

Sentences and models are also trivial to state.

```
Context (Σ : MachineSignature).                                   5.4.3
Context (A : Mod[INS_FOPEQ] (evt_sig Σ)).
Let X  := vars (evt_sig Σ).
Let X' := vars' (evt_sig Σ).

Definition MachineSen : Type :=
  FOSen (SigExtension (evt_sig Σ) X') *
  (events Σ -> FOSen (SigExtension (evt_sig Σ) (X ⊕ X')))%type.

Definition MachineMod : Type :=
  Env X A * list (events Σ * (Env X A))%type.
```

The semantic interpretation relation is debatably simpler to read in Coq than the account given in DEFINITION 5.3.6. Judge for yourself.

```
Fixpoint interp_machine_tail                                      5.4.4
    (st : Env X A)
    (models : list (events Σ * (Env X A)))
    (machine : events Σ -> FOSen (SigExtension (evt_sig Σ) (X ⊕ X')))
    {struct models}
    : Prop :=
  match models with
  | [] => True
  | (e, st') :: rest =>
      AlgExpansion A
        (join_envs st (retract_env id{FOSig} unprime st'))
```

```
      ⊨ machine e
    ∧ interp_machine_tail st' rest machine
  end.

Definition interp_machine
    (model : MachineMod)
    (machine : MachineSen)
    : Prop :=
  AlgExpansion A (retract_env id{FOSig} unprime (fst model))
    ⊨ fst machine
  ∧ interp_machine_tail (fst model) (snd model) (snd machine).

End Machine.
```

Event morphisms have only one condition: they must preserve the ordering on statuses.

```
Record EventMorphism (Σ Σ' : MachineSignature) : Type := {        5.4.5
  event_mor :> events Σ -> events Σ' ;
  preserves_status_order : ∀ e, status e ≤ status (event_mor e) ;
}.

Definition MachineSigMor (Σ Σ' : MachineSignature) :=
  ((Σ ~> Σ') * (EventMorphism Σ' Σ))%type.
```

The remaining constructions and proofs for *MacEVT* are not of any technical interest. We apply the same techniques we did for the previous institutions and we use the same basic lemmas we devised for *ACT*. The proof of satisfaction for machines is at Institutions/Machine.v#L287. The proof is long, but it's not complicated, though it could stand to be greatly simplified.

Let's press on and encode *LTL* in Coq.

5.5 *LTL in Coq*

LTL signatures, signature morphisms, models, and sentences, are all easy to define.

```
Record LTL_Signature : Type := {                                    5.5.1
  base :> FOSig ;
  vars : Vars base ;
}.

Record LTL_SigMor (Σ Σ' : LTL_Signature) : Type := {
  on_base :> Σ ~> Σ' ;
  on_vars : var_morphism on_base (vars Σ) (vars Σ') ;
}.

Record LTL_Model (Σ : LTLSig) := {
  base_alg : Mod[INS_FOPEQ] Σ ;
  trace : list (Env (vars Σ) base_alg) ;
}.

Inductive LTLSentence : Type :=
| FOLSen : FOSen (SigExtension Σ (vars Σ)) -> LTLSentence
| Or : LTLSentence -> LTLSentence -> LTLSentence
| Not : LTLSentence -> LTLSentence
| Next : LTLSentence -> LTLSentence
| Until : LTLSentence -> LTLSentence -> LTLSentence.
```

The basic category-theoretic proofs can be found at Institutions/LTL.v. As with *MacEVT*, repeating the proofs here would be wasteful of space.

We also defined a library for non-empty lists, defined inductively as follows.

```
Inductive NEList (A : Type) : Type :=                               5.5.2
| Last : A -> NEList A
| Cons : A -> NEList A -> NEList A.
```

There are many ways to represent non-empty lists—this is the most direct, but unfortunately requires us to redefine a basic list theory for non-empty lists (see Core/NEList.v).

To state the semantic interpretation function, we need to compute all 'tails' of a trace. The function tails defined below does this—for example, tails $[1, 2, 3]$ would be $[[1, 2, 3], [2, 3], [3]]$.

```
Fixpoint tails [A : Type] (ℓ : NEList A) : NEList (NEList A) :=     5.5.3
  match ℓ with
```

```
  | Last a => Last (Last a)
  | Cons x xs => Cons ℓ (tails xs)
  end.
```

The function tails satisfies the following relationship with map:

```
Lemma tails_map [A B] (f : A -> B) ℓ :                                  5.5.4
  tails (map f ℓ) = map (map f) (tails ℓ).
```

Finally, we need to be able to partition the trace at a natural number index to define the semantic interpretation for the until operator. This is unproblematic for lists since (bounded) natural numbers correspond naturally to positions in a list. (A natural number n is equivalent to a list of units of length n.) For non-empty lists, the functions must take at least one element and leave behind at least one element.

```
Fixpoint firstn [A] (n : nat) (ℓ : NEList A) : NEList A :=              5.5.5
  match n with
  | 0 | 1 =>
      match ℓ with
      | Last x => Last x
      | Cons x _ => Last x
      end
  | S n0 =>
      match ℓ with
      | Last x => Last x
      | Cons a ℓ0 => Cons a (firstn n0 ℓ0)
      end
  end.

Fixpoint skipn [A] (n : nat) (ℓ : NEList A) : NEList A :=
  match n with
  | 0 => ℓ
  | S n0 =>
      match ℓ with
      | Last x => Last x
      | Cons _ ℓ0 => skipn n0 ℓ0
      end
  end.
```

We can prove that under the condition $1 \leq n < \mathsf{length}(\ell)$, these two functions partition the list. We will use these functions exclusively under this condition.

```
Lemma partitions [A] (ℓ : NEList A) (n : nat) :                    5.5.6
  1 ≤ n < length ℓ -> ℓ = firstn n ℓ ++ skipn n ℓ.
```

To ensure the semantic interpretation function is correct, we will distinguish the case where β holds immediately and then handle the case where α holds in at least one state and β thereafter, since that allows us to isolate the case where the trace is partitioned into two non-empty lists.

Now we can define the semantic interpretation function.

```
Fixpoint interp_LTL_aux [Σ : LTLSig] (A : Mod[INS_FOPEQ] Σ)        5.5.7
    (π : NEList (Env (vars Σ) A))
    (ψ : LTLSen Σ) {struct ψ} : Prop :=
  match ψ with
  | FOLSen α => AlgExpansion A (NEList.hd π) ⊨ α
  | Or α β => interp_LTL_aux A π α ∨ interp_LTL_aux A π β
  | Not α => ¬ interp_LTL_aux A π α
  | Next α =>
      match π with
      | Last _ => True
      | Cons _ rest => interp_LTL_aux A rest α
      end
  | Until α β =>
    NEList.Forall (λ πⁱ, interp_LTL_aux A πⁱ α) (NEList.tails π)
    ∨ interp_LTL_aux A π β
    ∨ (∃ (n : nat),
         1 ≤ n < NEList.length π ∧
         interp_LTL_aux A (NEList.skipn n π) β ∧
         NEList.Forall (λ πⁱ, interp_LTL_aux A πⁱ α)
                       (NEList.tails (NEList.firstn n π)))
  end.
```

Compare this with the mathematical definition, DEFINITION 5.2.5.

5.5.1 *The satisfaction condition for LTL*

Surprisingly few additional facts are required to prove the satisfaction condition for *LTL*. The main lemmas revolve around the functions firstn and skipn, used in LISTING 5.5.7 to define *LTL*'s semantic interpretation function. Both commute with map f:

```
Lemma firstn_map [A B] (f : A -> B) n ℓ :                    5.5.8
  firstn n (map f ℓ) = map f (firstn n ℓ).

Lemma skipn_map [A B] (f : A -> B) n ℓ :
  skipn n (map f ℓ) = map f (skipn n ℓ).
```

The only other interesting lemmas involve properties of quantification over list items. We encourage you to take a look at the proof yourself in Institutions/LTL.v.

With *MacEVT* and *LTL* now defined, we can explain how to combine them using a duplex construction.

5.6 *Duplex Combination of LTL and MacEVT*

The type of combination we'll describe here is very simple in nature, but requires some technical definitions.

Recall first the definitions of functor (DEFINITION 2.1.2) and natural transformation (DEFINITION 2.1.3). We need to define *institution semi-morphisms* and *duplex institutions*. Since we're working with multiple institutions, standard institution constructions will be indexed by the particular institution. Both of the following definitions appear in CHAPTER 6 but we copy them here.

6.3.1 *Definition* [ST11]. Let $\mathcal{I}$ and $\mathcal{J}$ be institutions. An *institution semi-morphism* $\mu : \mathcal{I} \to \mathcal{J}$ consists of a functor $\mu^{\mathsf{Sig}} : \mathsf{Sig}_{\mathcal{I}} \to \mathsf{Sig}_{\mathcal{J}}$ and a natural transformation $\mu^{\mathsf{Mod}} : \mathsf{Mod}_{\mathcal{I}} \Rightarrow \mathsf{Mod}_{\mathcal{J}} \circ \mu^{\mathsf{Sig}}$.

6.3.2 *Definition* [ST11]. Given an institution semi-morphism $\mu : \mathcal{I} \to \mathcal{J}$, a *duplex institution* $\mathcal{I}$ plus $\mathcal{J}$ via μ has the following components:

- Signatures are those from $\mathcal{F}$.
- Given $\Sigma \in \mathrm{Sig}_{\mathcal{F}}$, sentences are either Σ-sentences in $\mathcal{F}$, or they are $\mu(\Sigma)$-sentences in $\mathcal{J}$, with the latter sometimes written as ψ via μ.
- Models are those from $\mathcal{F}$.
- The satisfaction relation is defined to be $M \models^{\mathcal{F}} \varphi$ for $\mathcal{F}$-sentences φ, and $\mu(M) \models^{\mathcal{J}}_{\mu(\Sigma)} \psi$ for ψ via μ.

The duplex construction comes 'for free' once we have defined the appropriate institution semi-morphism. This is part of the appeal of the duplex construction.

Institution semi-morphisms and duplex institutions are simple to define in Coq, and we will do so in SECTION 6.3, specifically LISTING 6.3.1. We will not repeat the Coq encoding here.

In this context, we should think of a duplex construction as a non-interacting 'sum' of two institutions—though it is a little different from a sum in the sense that the left- and right-hand sides of the construction play different roles. Let's consider the case of Event-B and LTL. Both machines and LTL sentences can be checked against machine traces, the former requiring a trace interleaving states and event names, and the latter requiring only states. There is a clear intuition that we should be able to use both *MacEVT* machines and *LTL* sentences to constrain the same trace—only the *LTL* sentences will not pay any attention to the event names. This syntactical mediation is facilitated by an institution semi-morphism from *MacEVT* to *LTL*. The functor taking *MacEVT* signatures to *LTL* signatures forgets the event names, and the natural transformation of models removes the event names from the trace. The functor is defined as follows.

```
Definition MacEVT2LTL_Sig : Sig[MacEVT] ⟶ Sig[LTL].                5.6.1
  unshelve refine {|
    fobj Σ := {|
      LTL.base := base (evt_sig Σ) ;
      LTL.vars := vars (evt_sig Σ) ;
    |} ;
    fmap := λ A B σ, _ ;
  |}; cbn in *; repeat intro.
```

```
  (* construct fmap *)
  - unshelve esplit.
    + exact (on_base (fst σ)).
    + exact (@on_vars _ _ (fst σ)).
  (* omitted proofs *)
Defined.
```

The natural transformation is defined as follows.

```
Definition MacEVT2LTL_Mod :                                          5.6.2
    Mod[MacEVT] ⟹ Mod[LTL] ∘ MacEVT2LTL_Sig^op.
  unshelve esplit; repeat intro.
  - unshelve esplit; intros.
    (* define the algebra and trace mapping *)
    + unshelve esplit; cbn in *.
      * exact (`1 H).
      * exact (fst (`2 H) :: map snd (snd (`2 H))).
    + exact f.
    (* omitted proofs *)
Defined.
```

And thus we have our duplex institution *EvtLTL*.

```
Definition EvtLTL_semi : InsSemiMorphism MacEVT LTL := {|            5.6.3
  μs_sig := MacEVT2LTL_Sig ;
  μs_mod := MacEVT2LTL_Mod ;
|}.

Definition EvtLTL : Institution :=
  Duplex MacEVT LTL EvtLTL_semi.
```

To forestall any premature celebrations, we should see if this institution really captures what we intended.

5.7 *Evaluation of EvtLTL*

Let's consider a quick example to see what we're working with. We will adapt the cars-on-a-bridge example from the Event-B manual.[12] Let's suppose we're working with a signature for natural numbers with the usual interpretation containing whatever constants and operations we

happen to need—let's say any amount of natural number literals, addition $(+)$, and subtraction $(-)$ for now. Let $\{d : \mathsf{nat}, n : \mathsf{nat}\}$ be the state variables.[‡] The initial event is $n := 0$. The other two events are

$$\begin{aligned}\mathsf{inc} &\triangleq \mathsf{when}\ n < d\ \mathsf{then}\ n := n + 1\\ \mathsf{dec} &\triangleq \mathsf{when}\ n > 0\ \mathsf{then}\ n := n - 1\end{aligned}$$

This can be written as the following *MacEVT* sentence:

$$\begin{aligned}mac \triangleq \langle n = 0, \{&\mathsf{inc} \mapsto (n < d) \wedge (n' = n + 1),\\ &\mathsf{dec} \mapsto (n > 0) \wedge (n' = n - 1)\}\rangle\end{aligned}$$

An example (finite) trace that models this sentence would be

$$\begin{aligned}tr \triangleq \langle \{n \mapsto 0, d \mapsto 3\}, (&(\mathsf{inc}, \{n \mapsto 1, d \mapsto 3\}),\\ &(\mathsf{inc}, \{n \mapsto 2, d \mapsto 3\}),\\ &(\mathsf{dec}, \{n \mapsto 1, d \mapsto 3\}),\\ &(\mathsf{dec}, \{n \mapsto 0, d \mapsto 3\}))\rangle\end{aligned}$$

To check that $tr \vDash mac$, we just need to prove the following proposition.

$$\begin{aligned}&(0 = 0)\\ \wedge\ &(0 < 3) \wedge (1 = 0 + 1)\\ \wedge\ &(1 < 3) \wedge (2 = 1 + 1)\\ \wedge\ &(2 > 0) \wedge (1 = 2 - 1)\\ \wedge\ &(1 > 0) \wedge (0 = 1 + 1)\end{aligned}$$

A big problem, however, is that this machine is missing its invariant, which is

$$0 \leq n < d$$

We can add this invariant by additionally asserting the *LTL*-sentence

$$inv \triangleq \mathsf{G}(0 \leq n < d)\ \mathsf{via}\ \mu$$

[‡]Actually in the original example d is a constant symbol and n is a state variable. We can add d to the first-order signature to enforce this distinction, but it's not particularly relevant for the example.

using the institution semi-morphism $\mu : MacEVT \to LTL$ constructed in the last section. Thus the presentation $\{mac, inv\}$ forms the machine and its invariant. We can get fancier and assert that n becomes 3 eventually, $\mathsf{F}(n = 3)$, or that n becomes 0 infinitely often, $\mathsf{G}\,\mathsf{F}(n = 0)$, and so on.

But the sky is sadly not the limit. Recall the characteristic liveness property (5.1) that we were interested in,

$$\mathsf{G}([\mathsf{req}] \to \mathsf{F}[\mathsf{ack}])$$

meaning that, whenever the req event happens, the ack event will eventually happen. This is not possible to express because $MacEVT$ and LTL are not really 'combined' in an interesting way—they are merely adjacent. LTL sentences can't mention events because LTL does not know anything about events.

To solve this problem, we could create a new institution that combines the sentences of LTL and Event-B directly and prove its satisfaction condition, but this seems like more work than necessary. Another simpler solution is to create a new version of LTL and tune it to work well with $MacEVT$. We'll discuss this solution now.

5.8 *LTL with Labels*

Thai Son Hoang *et al.* in [Hoa+16] indicate a grammar for LTL, the same one presented by Daniel Plagge and Michael Leuschel in [PL10] and which is used by the ProB tool.

$$\phi ::= \mathsf{true} \mid [e] \mid \mathsf{enabled}(e) \mid \neg\phi \mid \phi \wedge \phi \mid \phi\ \mathsf{U}\ \phi \mid \mathsf{X}\,\phi$$

This grammar adds two sentences to the usual list—$[e]$, meaning that the event e just happened in the current state, and $\mathsf{enabled}(e)$, meaning that the event e is enabled (i.e. its guard is true) in the current state.

We will modify this grammar to make sure everything works in the context we're employing it. In particular, we will replace the mention of events E with *event predicates* $E \to \mathsf{Prop}$ and we will not for now consider enabledness of events, since it requires some heavier machinery—

specifically, passing an event name's guard (a *FOPEQ* sentence) through the signatures and pairing them with labels.

The reasoning for replacing event names with event-name predicates will be discussed in SECTION 5.9, but for now we will just show that this works and enables the right sort of duplex construction. The institution we will construct is called *LLTL*. It is exactly the same as *LTL* except in the respects to follow.

The signatures for *LLTL* add labels to *LTL* signatures, which will be a type with decidable equality.

```
Record LLTL_Signature := {                                    5.8.1
  base :> FOSig ;
  vars : Vars base ;
  labels : DecType ;
}.
```

We will add the following sentence to the grammar of *LTL*.

```
Executed : (labels Σ -> bool) -> LLTLSentence                 5.8.2
```

The argument is a decidable label predicate. Probably the most common predicates will check if a particular event or set of events occurred. (But any predicate will work, of course.)

```
Fixpoint one_of (ℓ : list (labels Σ)) : labels Σ -> bool :=   5.8.3
  λ l', match ℓ with
  | [] => false
  | l :: rest =>
      if `2 (labels Σ) l l'
        then true
        else one_of rest l
  end.

Definition only (l : labels Σ) : labels Σ -> bool :=
  one_of [l]
```

In this case, the example sentence (5.1) would be written

$$\mathsf{G}([\mathsf{only}(\mathsf{req})] \rightarrow \mathsf{F}([\mathsf{only}(\mathsf{ack})]))$$

The satisfaction condition for $LLTL$ is exactly the same as for LTL—the additional case is trivial.

This institution can be combined with $MacEVT$ using a duplex construction in much the same way described in SECTION 5.6. The only difference is that event names are not dropped from the models and are instead 'mapped to' labels. Since we found it useful to require $LLTL$ labels to have decidable equality, we would need to enforce decidable equality for event names as well.

5.9 *An Extended Footnote: The Problem of Data Variance*

In Farrell's EVT, sentences were pairs of event names and first-order sentences. There's no *prima facie* reason why we could not have done the same—in fact, that was the original plan. But there are problems with this plan.

Let's suppose we want to model a set m of sentences of the form $\langle e, \phi \rangle$ against a machine trace. There are at least two problematic syntactical possibilities—

1. What happens if m contains two sentences $\langle e, \phi_1 \rangle$ and $\langle e, \phi_2 \rangle$ with the same event name?
2. What happens if m contains no sentence of the form $\langle e, \phi \rangle$ for some particular event e?

This is a particular instance of a common situation, where syntactically distinct things are not really distinct semantically. We would normally want to say that the first instance is the same as $\langle e, \phi_1 \wedge \phi_2 \rangle$ and that the second is $\langle e, \mathsf{true} \rangle$. But it's generally not a good idea to represent something permissively and posit *ad hoc* equivalences between distinct syntactical things. This is sometimes unavoidable—see SECTION 6.4 for a discussion of what to do in such cases—but not here.

There are at least two options. Let's start with the worse one: contorting the semantic interpretation relation to ignore syntactical differences. We can cover both of the above possibilities in one stroke without changing the syntax by iterating though the trace π and asking at each π_i

if

$$\bigvee_{(e,\phi)\in m} (e = \mathsf{fst}\ \pi_i) \wedge A^{\mathsf{snd}\ \pi_i} \vDash \phi \qquad (*)$$

This works, but it's clumsy.

The second and better solution is to fix the syntax. Notice that what we are describing with the conditions above is precisely a *correspondence* between event names and their actions and guards—i.e. a total function from event names to sentences. This ensures that each event name is accounted for (by the definition of 'total'), resolving the first problem, and that each event name has exactly one associated sentence that we can retrieve by simple function application (by the definition of 'function'), resolving the second problem, without the need for the somewhat silly disjunction $(*)$. This is one reason why we choose to represent machines as functions from event names to sentences.

The other reason relates to an even more fundamental problem with the 'machines as sets of sentences' conception: Models can no longer be traces $(\langle e_i, \phi_i \rangle)_i$. Why? Because the event names would appear in covariant position in both the sentences and the models, requiring an intolerably strict notion of signature morphism.

What do we mean by 'covariant position'? The notion of variance drawn upon here is akin to the concept of variance in category theory, but also akin to subtyping relations in programming language theory. If a type constructor is a functor of some description, we can ask how arrows in the source category are lifted to arrows in the target category—does the functor preserve or reverse the direction of arrows? We'll say that a type constructor T is covariant if arrows $a \to b$ are lifted to arrows $T_a \to T_b$, and contravariant if the same arrow is lifted to $T_b \to T_a$. For example, the list type is covariant because arrows $f : a \to b$ are lifted to $\mathsf{map}\ f : \mathsf{List}(a) \to \mathsf{List}(b)$, but the type constructor $\lambda\, t.\ t \to c$ is contravariant since $f : a \to b$ would be lifted to precomposition

$$\lambda\, g.\ g \circ f : (b \to c) \to (a \to c)$$

as we have seen many times in the model functors we have discussed. In type constructors with multiple parameters (like $(\to)$, the arrow type)

the type might be covariant in one parameter but contravariant in another. To distinguish these cases, we describe the parameter as respectively appearing in covariant or contravariant position.

Back to the problem at hand: A signature morphism $f : \sigma \to \tau$ operating on τ-models must be able to map τ-states to σ-states—which it may do by retraction—but also τ-events to σ-events, which it can only do if the mapping on event names is reversed with respect to the direction of the signature morphism. This precludes the use of sets of pairs $\langle e, \phi \rangle$ for *MacEVT* sentences unless the signature morphism can also map σ-events to τ-events. Allowing both requires the signature morphism to be a bijection on event names, which is far too strong a requirement. In general, the types of the sentences and models must have opposing variance in the position of the event name type.

The problem is that when we try to combine *MacEVT* and *LLTL*, if we were to stick with the sentences $[e]$ and $\mathsf{enabled}(e)$, we run into precisely the same problem again. That's why we replace the event names in *LLTL* with event-name predicates—it places event names appearing in the sentences in opposing variance to their position in the models.

We've so far introduced and formally encoded six institutions: *FOPEQ*, *ACT*, *LTL*, *LLTL*, *MacEVT*, and *EvtLTL*. The relationships between these institutions are apparent. But we have not so far developed many tools to relate them, nor have we made any attempt to show that they satisfy interesting or useful general properties of logical systems. That will be the subject of the next chapter.

SIX

Institution-Independent Constructions

Our focus up to now has been on constructing specific key institutions. Besides the duplex construction in CHAPTER 5 we have scarcely hinted at institution-independent constructions. We will devote our attention to such constructions in this chapter.

What we call an 'institution-independent' construction is any construction in which we speak of an arbitrary institution $\mathcal{I}$, possibly satisfying so-and-so properties. It is here that we take the top-down approach of universal logic: What is it we can say about institutions in general? What general constructions are possible? Can we encode them in Coq?

We'll begin by proving some simple facts about institutions and semantic consequence in SECTION 6.1. Then we will encode institution morphisms in SECTION 6.2 and duplex institutions in SECTION 6.3. Next we discuss the amalgamation property in SECTION 6.4, entailment systems for institutions in SECTION 6.5, and some foundation-independent logics in SECTION 6.6.

Most of the code for this chapter can be found in Institutions.v.

6.1 *Institution-Independent Model Theory*

In this section, we'll prove a simple but important result—that semantic consequence is preserved by signature translation.

6.1.1 *Theorem* [ST11]. Let $\mathcal{I}$ be an institution and let $f : \sigma \to \tau$ be a signature morphism in $\mathcal{I}$. Let Ψ be a set of $\mathcal{I}$-sentences and let ψ be an

$\mathcal{J}$-sentence. Then

$$\Psi \models \psi \quad \text{implies} \quad f(\Psi) \models f(\psi)$$

The symbol '$\models$' denotes *semantic consequence* and roughly means that the sentence ψ 'follows from' Ψ. We will make this notion precise in Coq, but first we will need to encode some basic set theory and model theory.

6.1.1 *Set theory in Coq*

We'll rely on the Coq.Sets.Ensembles definition for sets, which regards a set of elements of type T as a predicate $T \to$ Prop, as we discussed in SECTION 2.2. To that we will add the following definitions—the notation should be self-explanatory.

```
Context [X Y : Type].                                                    6.1.1

Definition set_preimage (f : X -> Y) (S : SetOf Y) : SetOf X :=
  { x : X // f x ∈ S }.

Definition set_image (f : X -> Y) (S : SetOf X) : SetOf Y :=
  { y : Y // ∃ x : X, x ∈ S ∧ f x = y }.

Notation "f '⁻¹'" :=
  (set_preimage f) (at level 5, format "f ⁻¹'") : sets_scope.
```

We adopt the axiom of extensional equality of sets and present it in the following more useful form.

```
Theorem set_ext (S T : SetOf X) :                                        6.1.2
  (∀ x, x ∈ S <-> x ∈ T) -> S = T.
Proof.
  intros H.
  apply Extensionality_Ensembles.
  split; intros ? ?; apply H; auto.
Qed.
```

We also prove the following useful theorem.

```
Theorem set_mem_preimage (f : X -> Y) S a :                    6.1.3
  a ∈ f⁻¹' S <-> f a ∈ S.
Proof. firstorder. Qed.
```

That is all the set theory we will need for this chapter.

6.1.2 *Basic model theory*

With some set-theoretic basics defined, we fix an institution $\mathcal{I}$ and define some very basic concepts in model theory.

6.1.2 *Definition* [ST11]. A Σ-*presentation* in $\mathcal{I}$ for some $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$ is a pair $\langle \Sigma, \Phi \rangle$, where Φ is a set of Σ-sentences.

Presentations pair off the signature with the set of sentences, but for the rest of this section we will work with the set of sentences only (since the signature will be fixed). We will occasionally still refer to this as a presentation.

6.1.3 *Definition* [ST11]. Let σ be a signature.

- Let Φ be a set of σ-sentences. The *models of* Φ, denoted $Mod(\Phi)$, is a set consisting of all σ-models satisfying all sentences in Φ.
- Let $\mathcal{M}$ be a set of σ-models. The *theory of* $\mathcal{M}$, denoted $Th(\mathcal{M})$, is a set consisting of all σ-sentences satisfied by all models in $\mathcal{M}$.
- The *closure* of set of σ-sentences Φ is $Cl(\Phi) = Th(Mod(\Phi))$.
- The *closure* of a set of σ-models $\mathcal{M}$ is $Cl(\mathcal{M}) = Mod(Th(\mathcal{M}))$.
- A set of sentences Φ is *closed* if $\Phi = Cl(\Phi)$.

Note that the sentence closure $Cl(\Phi)$ contains all sentences that follow semantically from the sentences in Φ. This concept is used to define semantic consequence.

6.1.4 *Definition* [ST11]. A sentence φ is a *semantic consequence* of a set of sentences Φ, written $\Phi \models \varphi$, if $\varphi \in Cl(\Phi)$.

Let's define semantic consequence in Coq. We will encode everything in DEFINITIONS 6.1.3 & 6.1.4 in order of appearance. The definitions above and in Coq should be compared directly.

```
Context [I : Institution] [σ : Sig].                                    6.1.4

Definition presentation σ := (SetOf (Sen σ)).
Definition model_class σ := (SetOf (Mod σ)).

Definition modelsof (Ψ : presentation σ) : SetOf (Mod σ) :=
  { m : Mod σ // ∀ ψ, ψ ∈ Ψ -> m ⊨ ψ }.

Definition theoryof (M : model_class σ) : SetOf (Sen σ) :=
  { φ : Sen σ // ∀ m, m ∈ M -> m ⊨ φ }.

Definition closure_sen (Ψ : presentation σ) :=
  theoryof (modelsof Ψ).

Definition closure_mod (M : model_class σ) :=
  modelsof (theoryof M).

Definition semantic_consequence [σ]
    (Φ : presentation σ) (φ : Sen σ) :=
  φ ∈ closure_sen Φ.

Definition closed (Ψ : presentation σ) :=
  Ψ = closure_sen Ψ.

Local Infix "⇒" := semantic_consequence.
```

Now we can prove that signature translation preserves semantic consequence. Let's first indicate a proof on paper. An alternative proof can be found in [ST11, §4.2], Proposition 4.2.9.

Proof of THEOREM 6.1.1. By definition, $f(\Phi) \Vdash f(\phi)$ is true if, for all $M \in Mod(f(\Phi))$, we have $M \models f(\varphi)$. So, (1) Let $M \in Mod(f(\Phi))$. We aim to prove $M \models f(\varphi)$. (2) By satisfaction this is equivalent to proving $M|_f \models \varphi$. (3) Since $\Phi \Vdash \varphi$, the foregoing is true if $M|_f \in Mod(\Phi)$. (4) By definition, $M|_f \in Mod(\Phi)$ is true if and only if for all $\psi \in \Phi$, we have $M|_f \models \psi$, (5) which by satisfaction is equivalent

to $M \models f(\psi)$. (6) Since by assumption$^{(1)}$ $M \in Mod(f(\Phi))$, we are done. □

The encoding in Coq corresponds exactly with the handwritten proof above. The steps above have been linked with the roughly equivalent reasoning step in Coq.

```
Lemma preserves_consequence                                    6.1.5
    (f : σ ~> τ) (Φ : presentation σ) (φ : Sen σ) :
  Φ ⇒ φ -> set_image (fmap[Sen] f) Φ ⇒ fmap[Sen] f φ.
Proof.
  (* 1 *) intros H m H1.
  (* 2 *) rewrite sat.
  (* 3 *) apply H.
  (* 4 *) intros ψ H2.
  (* 5 *) rewrite <- sat.
  (* 6 *) apply H1. exists ψ. auto.
Qed.
```

The proof involves only basic tactics and reasoning steps, and requires few overall background facts—some set theory (which is really predicate logic), some propositional reasoning, and the satisfaction condition. A custom tactic could automate this proof.

It also reasons backwards, which is not typical for handwritten proofs. My own mathematical education implicitly emphasised forward reasoning—working from assumptions to conclusion—since that is the natural direction in which reasoning is metaphorically conceptualised as 'going'. But proof assistants like Coq are better suited to backward reasoning, working from conclusion to assumptions: notice that all tactics in the above proof directly manipulate the goal and not the hypotheses. Coq supports both forms of reasoning, and we will see how to encode forward reasoning in Coq in just a moment.

First, note that preservation of consequence by a signature morphism f, as we have just proved, is equivalent to $f(Cl(\Phi)) \subseteq Cl(f(\Phi))$. This form is sometimes more useful and is encoded in Coq as follows.

```
Lemma alt_preserves_consequence                                6.1.6
    (f : σ ~> τ) (Φ : presentation σ) :
```

```
  set_image (fmap[Sen] f) (closure_sen Φ)
    ⊆ closure_sen (set_image (fmap[Sen] f) Φ).
```

We will use this version in the following and final example, where we will prove that whenever Φ is closed, $f^{-1}(\Phi)$ is closed too. This is Corollary 4.2.12 in [ST11, §4.2]—the proof below closely matches the proof given there (and shows how to conduct forward reasoning in Coq).

```
Lemma corollary_4_2_12 (f : σ ~> τ) (Φ' : presentation τ) :        6.1.7
  closed Φ' -> closed ((fmap[Sen] f)⁻¹ Φ').
Proof.
  (* suppose Φ' is closed … *)
  intros H. unfold closed in H.
  (* deal with implicit reverse case *)
  apply set_ext. intros φ. split. { apply closure_superset. }
  (* let φ ∈ Cl(f⁻¹(Φ')) … *)
  intros H'.
  (* first notice that … *)
  assert (hypo₁ :
    set_image (fmap[Sen] f) ((fmap[Sen] f)⁻¹' Φ') ⊆ Φ').
  { intros ψ H0. repeat destruct H0. rewrite <- H1. apply H0. }
  assert (hypo₂ :
    closure_sen (set_image (fmap[Sen] f) ((fmap[Sen] f)⁻¹' Φ'))
    ⊆ closure_sen Φ').
  { apply (closure_preserves_order _ _ hypo₁). }

  (* now, by proposition 4.2.9 (consequence preservation) *)
  assert (hypo₃ :
    fmap[Sen] f φ ∈
      closure_sen (set_image (fmap[Sen] f) ((fmap[Sen] f)⁻¹' Φ'))).
  {
    apply alt_preserves_consequence.
    rewrite <- set_mem_preimage.
    exists φ; auto.
  }
  rewrite <- H in hypo₂.
  assert (final : fmap[Sen] f φ ∈ Φ'). { apply hypo₂. assumption. }

  (* have f(φ) ∈ Φ'; hence φ ∈ f⁻¹(Φ') *)
  rewrite set_mem_preimage. exact final.
Qed.
```

The pattern is to assert intermediary claims and then to justify them with proofs in braces following them. (There's more detail about how this tactic works in the tactic index on page 135.)

The simplicity of these proofs gives us reason to believe that a comprehensive formalisation of the work in [ST11] could be only a matter of time. Note also that the proofs here are explicit for demonstrative purposes. It seems likely that an automated prover—like CoqHammer [CK18] utilising a first-order prover such as Vampire [RV02] or cvc5 [Bar+22]—could discharge many of these proofs automatically.

6.2 *Institution Morphisms*

There are three broad kinds of institution morphisms: semi-morphisms, morphisms, and comorphisms. These morphisms are crucial for *heterogeneous specification*, in which multiple logics are used in tandem to specify a system.

- Institution semi-morphisms are used to define duplex institutions, as we have seen already in CHAPTER 5 and which we will see again in the next section.
- Institution morphisms explain how a more complex institution builds on a simpler one.
- Institution comorphisms explain how to encode the sentences of a more complex institution in a simpler one; or alternatively they can express a kind of institution inclusion.

All of these concepts are defined in [ST11, Chapter 10]. It is straightforward to give full definitions for these concepts in Coq, and the Coq definitions mirror exactly the mathematical definitions. Recall that functors use '⟶', while natural transformations use '⟹'.

```
Record InsSemiMorphism (I I' : Institution) := {                    6.2.1
  μs_sig : Sig[I] ⟶ Sig[I'] ;
  μs_mod : Mod[I] ⟹ Mod[I'] ∘ μs_sig^op
}.

Record InsMorphism (I I' : Institution) := {
  μ_sig :> Sig[I] ⟶ Sig[I'] ;
  μ_sen :  Sen[I'] ∘ μ_sig ⟹ Sen[I] ;
  μ_mod :  Mod[I] ⟹ Mod[I'] ∘ μ_sig^op ;
  μ_sat : ∀ Σ M ψ',
    M ⊨ μ_sen Σ ψ' <-> μ_mod Σ M ⊨ ψ'
}.

Record InsComorphism (I I' : Institution) := {
  ρ_sig :> Sig[I] ⟶ Sig[I'] ;
  ρ_sen :  Sen[I] ⟹ Sen[I'] ∘ ρ_sig ;
  ρ_mod :  Mod[I'] ∘ ρ_sig^op ⟹ Mod[I] ;
  ρ_sat : ∀ Σ M' φ,
    ρ_mod Σ M' ⊨ φ <-> M' ⊨ ρ_sen Σ φ
}.
```

We've encoded an institution semi-morphism already in SECTION 5.6. We could in the future relate *FOPEQ* and *ACT* directly via an institution comorphism inclusion, however another (perhaps better) idea would be to show formally the straightforward fact that *FOPEQ* 'encodes' *ACT* via a comorphism $ACT \to FOPEQ$. Institutions which can be formally encoded in first-order logic, in a manner that is semantics-preserving, would have access to a wide range of automatic theorem provers. Such a translation is often not direct, in which case we may need to encode institutions of theories [Dia22, §4.a] and theoroidal comorphisms.

6.3 *Duplex Institutions*

A duplex institution is probably the simplest kind of institution combination. It relies only on the existence of an institution semi-morphism between the two institutions.

6.3.1 *Definition* [ST11]. Let $\mathcal{I}$ and $\mathcal{J}$ be institutions. An *institution semi-morphism* $\mu : \mathcal{I} \to \mathcal{J}$ consists of a functor $\mu^{\mathsf{Sig}} : \mathsf{Sig}_{\mathcal{I}} \to \mathsf{Sig}_{\mathcal{J}}$ and a natural transformation $\mu^{\mathsf{Mod}} : \mathsf{Mod}_{\mathcal{I}} \Rightarrow \mathsf{Mod}_{\mathcal{J}} \circ \mu^{\mathsf{Sig}}$.

6.3.2 *Definition* [ST11]. Given an institution semi-morphism $\mu : \mathcal{I} \to \mathcal{J}$, a *duplex institution* $\mathcal{I}$ plus $\mathcal{J}$ via μ has the following components:

- Signatures are those from $\mathcal{I}$.
- Given $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$, sentences are either Σ-sentences in $\mathcal{I}$, or they are $\mu(\Sigma)$-sentences in $\mathcal{J}$, with the latter sometimes written as ψ via μ.
- Models are those from $\mathcal{I}$.
- The satisfaction relation is defined to be $M \models^{\mathcal{I}} \varphi$ for $\mathcal{I}$-sentences φ, and $\mu(M) \models^{\mathcal{J}}_{\mu(\Sigma)} \psi$ for ψ via μ.

The idea of a duplex institution is simple: If you have two separate institutions with similar signatures and models, then specifications can intuitively include sentences from both institutions side-by-side. The 'similarity' of the signatures and models is captured by an institution semi-morphism between the two institutions.

For this definition to make sense, the satisfaction condition must hold. This is left as an exercise for the reader in [ST11], but for those readers who have made it this far, we will spoil the fun by proving it in Coq.

```
Definition Duplex [I I']                                        6.3.1
    (μ : InsSemiMorphism I I') : Institution.
  unshelve refine {|
    Sig := Sig[I] ;
    Sen := CoprodSetFunctor Sen[I] (Sen[I'] ∘ μs_sig μ) ;
    Mod := Mod[I] ;
    interp Σ M ψ :=
      match ψ with
      | inl ψ => M ⊨ ψ
      | inr ψ => μs_mod μ Σ M ⊨ ψ
      end ;
  |}; repeat intro.
  destruct φ; cbn.
  - apply sat.
```

```
  - now rewrite <- naturality, sat.
Defined.
```

The proof follows from the naturality of the natural transformation μ^{Mod} and satisfaction for the base institutions.

Given two institutions and an institution semi-morphism between them, we get a duplex construction for free. We used this in CHAPTER 5 to combine an institution for LTL with an institution for Event-B, both given a trace semantics. See in particular SECTION 5.6.

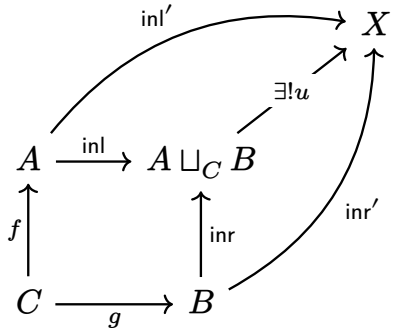
6.4 *Properties of Institutions*

Most concrete institutions are not merely institutions, but also satisfy some interesting or useful properties that can be expressed abstractly using the language of institution theory. We are particularly interested in the *amalgamation property*. But before defining amalgamation, we will need to define pushouts.

6.4.1 *Definition* [Awo10]. The *pushout* of arrows $f : C \to A$ and $g : C \to B$ is an object denoted $A \sqcup_C B$ with two arrows

$$\mathsf{inl} : A \to A \sqcup_C B$$
$$\mathsf{inr} : B \to A \sqcup_C B$$

such that $\mathsf{inl} \circ f = \mathsf{inr} \circ g$ and such that $A \sqcup_C B$ is universal with this property—that is, given another object X and arrows $\mathsf{inl}' : X \to A$ and $\mathsf{inr}' : X \to B$ satisfying the same condition, there is a unique arrow $u : A \sqcup_C B \to X$ such that $\mathsf{inl}' = u \circ \mathsf{inl}$ and $\mathsf{inr}' = u \circ \mathsf{inr}$. This situation can be summarised by the following diagram.



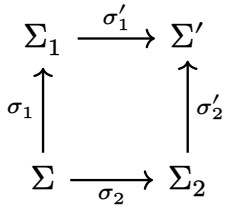
Such universality conditions identify objects up to unique isomorphism—i.e. the arrow $u : A \sqcup_C B \to X$ above is an isomorphism. This means that once we construct a pushout of two objects, we know that the constructed pushout is the unique such pushout up to isomorphism.

In Set, pushouts are quotients of disjoint unions—suppose there are three sets A, B, and C; the pushout of functions $f : C \to A$ and $g : C \to B$ is the quotient $(A + B)/\!\sim$, where $\mathsf{inl}(f(c)) \sim \mathsf{inr}(g(c))$ for all $c \in C$, and where $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ are the standard inclusions. In the situation where C is regarded as common to A and B (perhaps even $C = A \cap B$) then the pushout identifies copies of elements of C that appear in the disjoint union. Regarded as topological spaces, this construction is like pasting two spaces together (or the same space to itself) along a subspace.

In our situation where we consider pushouts of signatures, C often plays the role of symbols 'common' to both A and B; the pushout identifies copies of those symbols appearing in the signatures A and B so that they do not appear twice in the union. This would be particularly undesirable here because the models would then be permitted to give different interpretations to the 'same' symbols.

We are now ready to define amalgamation.

6.4.2 *Definition* [ST11]. Let $\mathcal{I}$ be an institution. The Sig-diagram



admits amalgamation if

- for any Σ_1-model M_1 and Σ_2-model M_2 such that $M_1|_{\sigma_1} = M_2|_{\sigma_2}$, there is a unique Σ'-model M', called the *amalgamation* of M_1 and M_2, such that $M'|_{\sigma'_1} = M_1$ and $M'|_{\sigma'_2} = M_2$.
- for any Σ_1-model morphism $f_1 : M_{11} \to M_{12}$ and Σ_2-model morphism $f_2 : M_{21} \to M_{22}$ such that $f_1|_{\sigma_1} = f_2|_{\sigma_2}$, there is a unique Σ'-model morphism $f' : M'_1 \to M'_2$, called the *amalgamation* of f_1 and f_2, such that $f'|_{\sigma'_1} = f_1$ and $f'|_{\sigma'_2} = f_2$.

We further say that $\mathcal{I}$ *has the amalgamation property* if all pushouts in Sig exist and all pushout diagrams admit amalgamation.

A sufficient condition for an institution $\mathcal{I}$ to admit amalgamation is *semi-exactness*.

6.4.3 *Definition.* An institution $\mathcal{I}$ is *semi-exact* if all pushouts exist in $\mathsf{Sig}_{\mathcal{I}}$ and the model functor $\mathsf{Mod}_{\mathcal{I}} : \mathsf{Sig}_{\mathcal{I}}^{\mathsf{op}} \to \mathsf{Cat}$ preserves pushouts.

While we have yet to explore proving that the institutions we have defined satisfy the amalgamation property, part of our development was explicitly designed to support it. Recall in SECTION 3.3 that we represent indexed types not as $A : J \to \mathcal{U}$ but as a pair $\langle A : \mathcal{U}, \mathsf{tag} : A \to J \rangle$. This makes coproducts—and hence pushouts—substantially easier to construct.

```
Definition SigSum : FOSig := {|                                    6.4.1
  Sorts := Sorts A + Sorts B ;
  Funcs := {|
    tagged_data := Funcs A + Funcs B ;
    get_tag F :=
      match F with
```

```
      | inl F => (map inl (fst (get_tag F)), inl (snd (get_tag F)))
      | inr F => (map inr (fst (get_tag F)), inr (snd (get_tag F)))
      end ;
  |} ;
  Preds := {|
    tagged_data := Preds A + Preds B ;
    get_tag P :=
      match P with
      | inl P => map inl (get_tag P)
      | inr P => map inr (get_tag P)
      end ;
  |}
|}.
```

But computing pushouts seems somewhat more difficult. A pushout in type theory seems as if it must be the quotient type or setoid $(A+B)/{\sim}$ where $\mathsf{inl}(f(c)) \sim \mathsf{inr}(g(c))$.

In written mathematics one gets away with treating rather coarse equivalences as if they were true equalities; for example, we are permitted in virtually all cases to substitute the fraction $1/2$ for $2/4$ or vice versa, despite the fact that they are presented differently. Then we simply rule out nonsense operations on fractions like $f(p/q) = p+q$ which depend on the way in which the fraction is presented—that is, operations on fractions must be well-defined.

In type theory, if we encode $\mathbb{Q}$ in the usual way, as $\mathbb{Z}\times(\mathbb{Z}\setminus\{0\})$, then $1/2$ and $2/4$ are distinct objects that we nevertheless want to consider 'equivalent'. We cannot simply insist axiomatically that $1/2 = 2/4$—that is a contradiction—so instead we define an equivalence relation on $\mathbb{Q}$ and only worry about whether fractions are equivalent and whether operations on fractions are well-defined with respect to this equivalence. This seems no different from the set-theoretic encoding, but syntax and representation is important in type theory, and there is unfortunately a considerable cost in using such quotient types everywhere. Equality in type theory has a number of useful properties that arbitrary equivalences do not. For example, we lose general substitution—we cannot always write $1/2$ in place of $2/4$, unless we can explicitly prove that the relev-

ant context cannot distinguish them. We therefore do not choose this approach.

Alternatively, if Coq had native support for higher inductive types (HITs), we could define pushouts like this, in the following pseudo-Coq syntax [nLa23].

```
Inductive hpushout {A B C : Type}                                    6.4.2
  (f : C -> A) (g : C -> B) : Type :=
| inl : A -> hpushout f g
| inr : B -> hpushout f g
| glue : ∀ c : C, inl (f c) == inr (g c).
```

Fortunately, it is possible to approximate HITs in Coq using a 'hack' involving private inductive types.[13] Private inductives allow unsafe operations local to their defining module and prevent unsafe operations outside it, relying only on what the author has provided—generally a custom induction principle. The idea is to axiomatise the higher equivalences and provide a custom induction principle involving that equivalence, while preventing the user from doing standard case analysis and breaking things outside the module. That rules out common Coq tactics like induction, destruct, case, and so on. The following definition showcases the technique and might suffice for our purposes.

```
Private Inductive Pushout {A B C : Type}                             6.4.3
  (f : C -> A) (g : C -> B) : Type :=
| pinl : A -> Pushout f g
| pinr : B -> Pushout f g.

Arguments pinl {A B C f g}.
Arguments pinr {A B C f g}.

Axiom pglue :
  ∀ {A B C : Type}
    {f : C -> A} {g : C -> B}
    (c : C),
  pinl (f := f) (f c) = pinr (g := g) (g c).

Definition pushout_ind
  {A B C : Type} {f : C -> A} {g : C -> B}
```

```
  (P : Pushout f g -> Type)
  (u : ∀ a, P (pinl a))
  (v : ∀ b, P (pinr b))
  (p : ∀ c, rew pglue c in u (f c) = v (g c))
  : ∀ x, P x.
Proof. destruct x; auto. Defined.

Axiom pushout_ind_pglue :
  ∀ {A B C : Type} {f : C -> A} {g : C -> B}
    (P : Pushout f g -> Type)
    (u : ∀ a, P (pinl a))
    (v : ∀ b, P (pinr b))
    (p : ∀ c, rew pglue c in u (f c) = v (g c)),
  ∀ c, f_equal_dep _ (pushout_ind P u v p) (pglue c) = p c.
```

Not nearly as neat as LISTING 6.4.2, but it works. The axiom pglue is precisely the identity inl $\circ f$ = inr $\circ g$ from DEFINITION 6.4.1. Of course this is inconsistent with Coq if we allow typical reasoning employing disjointness of constructors, since for HITs constructors are often not disjoint—precisely the situation we are in.

Signature pushouts are simple to encode using this definition. In fact, the construction is nearly identical to the coproduct in LISTING 6.4.1.

```
Definition SigPushout : FOSig.                                    6.4.4
  refine {|
    Sorts := Pushout (on_sorts f) (on_sorts g) ;
    Funcs := {|
      tagged_data := Pushout (on_funcs f) (on_funcs g) ;
    |} ;
    Preds := {|
      tagged_data := Pushout (on_preds f) (on_preds g) ;
    |} ;
  |}.
  (* proofs omitted *)
Defined.
```

We omit the proofs, but they can be found at FOL/Amalgamation.v#L40. Defining amalgamation proper for *FOPEQ* involves explicitly constructing the amalgamated algebra, which we have not yet done—but once it is done, *ACT*, *EVT*, and *MacEVT* should soon follow. Evaluating this

approach against the more traditional setoid approach may be the subject of future work.

6.5 *Entailment Systems*

José Meseguer defines an entailment system for an arbitrary institution in [Mes89], and Răzvan Diaconescu gives a slightly different presentation of the same concept in [Dia22], which we will define here.

6.5.1 *Definition.* An *entailment relation* for an institution $\mathcal{I}$ is any binary relation $\vdash_\Sigma \subseteq \mathcal{P}(\mathsf{Sen}_\mathcal{I}(\Sigma)) \times \mathsf{Sen}_\mathcal{I}(\Sigma)$ satisfying the following conditions.

- If $\varphi \in \Gamma$ then $\Gamma \vdash_\Sigma \varphi$.
- If $\Gamma \vdash_\Sigma \psi$ for each $\psi \in \Gamma'$, and $\Gamma' \vdash_\Sigma \varphi$, then $\Gamma \vdash_\Sigma \varphi$.
- If $\Gamma \vdash_\Sigma \varphi$, then for any $\sigma : \Sigma \to \Sigma'$, $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(\varphi)$.

In Coq, entailment relations can be directly represented by the following class.

```
Context {I : Institution}.                                          6.5.1
Context (R : ∀ Σ : Sig[I], list (Sen[I] Σ) -> Sen[I] Σ -> Prop).
Local Notation "Γ ⊢ φ" := (R _ Γ φ) (at level 5).

Class EntailmentRel := {
  entailment_axiom_1 : ∀ Σ Γ (φ : Sen Σ), In φ Γ -> Γ ⊢ φ ;
  entailment_axiom_2 : ∀ Σ Γ Γ' (φ : Sen Σ),
    List.Forall (λ ψ, Γ ⊢ ψ) Γ' -> Γ' ⊢ φ -> Γ ⊢ φ ;
  entailment_axiom_3 : ∀ (Σ Σ' : Sig[I]) (σ : Σ ~> Σ') Γ φ,
    Γ ⊢ φ -> (map (fmap[Sen] σ) Γ) ⊢ (fmap[Sen] σ φ) ;
}.
```

Expressing specific entailment relations as an inductive type, with one constructor per proof rule, is a standard technique in Coq. Here's a subset of a homebrewed sequent calculus I've been using for testing. It is essentially the standard classical sequent calculus but with some extra rules added to make some proofs simpler to express.

```
Inductive FOPEQ_entails :                                          6.5.2
  list (FOPEQ A ctx) -> list (FOPEQ A ctx) -> Prop :=
| triviality  : ∀ Γ, Γ ⊢ []
| top_R       : ∀ Γ Δ, Γ ⊢ FOL_T :: Δ
| bot_L       : ∀ Γ Δ, FOL_F :: Γ ⊢ Δ
| hypothesis  : ∀ Γ Δ a, a :: Γ ⊢ a :: Δ
| reorder_hyp : ∀ Γ Δ a b, a :: b :: Γ ⊢ Δ -> b :: a :: Γ ⊢ Δ
| cycle_hyp   : ∀ Γ Δ a, Γ ++ [a] ⊢ Δ -> a :: Γ ⊢ Δ
| weakening   : ∀ Γ Δ a, Γ ⊢ Δ -> a :: Γ ⊢ Δ
| contr_l     : ∀ Γ Δ a, a :: a :: Γ ⊢ Δ -> a :: Γ ⊢ Δ
| contr_r     : ∀ Γ Δ a, Γ ⊢ a :: a :: Δ -> Γ ⊢ a :: Δ

| and_l1  : ∀ Γ Δ a b, a :: Γ ⊢ Δ -> And a b :: Γ ⊢ Δ
| and_l2  : ∀ Γ Δ a b, b :: Γ ⊢ Δ -> And a b :: Γ ⊢ Δ
| and_e1 : ∀ Γ Δ a b, Γ ⊢ And a b :: Δ -> Γ ⊢ a :: Δ
| and_e2 : ∀ Γ Δ a b, Γ ⊢ And a b :: Δ -> Γ ⊢ b :: Δ

| or_r1  : ∀ Γ Δ a b, Γ ⊢ a :: Δ -> Γ ⊢ Or a b :: Δ
| or_r2  : ∀ Γ Δ a b, Γ ⊢ b :: Δ -> Γ ⊢ Or a b :: Δ
...
    where "Γ ⊢ φ" := (FOPEQ_entails Γ φ).
```

Given an entailment system—whether or not we prove that it is an 'entailment system' as we have just defined it—we can easily express important properties like soundness. In first-order logic, this is the condition that for any set of sentences Γ, if $\Gamma \vdash \psi$ then $\Gamma \vDash \psi$, i.e. the entailment system proves only true statements with respect to the semantics. We can more-or-less directly write this in Coq.

```
Theorem fol_soundness `(Γ : list (Sen A)) (φ : Sen A) :            6.5.3
  Γ ⊢ φ -> list_to_set Γ ⇒ φ.
```

One significant advantage of the deep-encoding approach to this formalisation is that it easily enables us to express propositions of this kind.

It would be an interesting future project to prove this and related soundness conditions in Coq. Especially interesting would be a meta-theoretical correctness proof for Event-B's proof system with respect to its trace semantics, or a proof that Event-B's syntactical refinement mechanisms correspond to model-theoretic refinement.

6.6 *Foundation-Independent Logics*

In this section we'll show how to build a few different parameterised institutions, in which the 'underlying' institution is not specified.

6.6.1 *Foundation-Independent Linear-Temporal Logic*

The most common form of LTL, propositional LTL, is in some sense built 'over' propositional sentences. But the choice of propositional logic as a base is somewhat arbitrary—indeed, our version of LTL is built over first-order logic. Can we replace $FOPEQ$ by an arbitrary institution $\mathcal{I}$ in the definition of an institution for LTL? If so, how useful is the resulting construction?

We can use an arbitrary institution $\mathcal{I}$ as the ground logic for LTL, called $LTL(\mathcal{I})$. In this section we'll describe the construction and its formalisation in Coq. This construction appears in [ST11] and we construct it here to demonstrate the simplicity of the construction in the framework.

6.6.1 *Theorem.* Let $\mathcal{I}$ be an institution. There is an institution $LTL(\mathcal{I})$ with the following components.

- Signatures are those from $\mathcal{I}$.
- Models are $\mathbb{N} \to \mathsf{Mod}_{\mathcal{I}}(\sigma)$.
- Sentences are the usual LTL connectives.
- Semantic entailment is also as usual but defers to $\mathcal{I}$-entailment for ground sentences.

The idea is to pick a different $\mathcal{I}$-model per time step and interpret the ground sentences using that model.

Let's be more precise about these constructions. The sentences are defined as usual.

```
Context [I : Institution].                                    6.6.1

Inductive fobj_LTL (Σ : Sig[I]) : Type :=
```

```
| ltl_sen   : Sen[I] Σ -> fobj_LTL
| ltl_true  : fobj_LTL
| ltl_conj  : fobj_LTL -> fobj_LTL -> fobj_LTL
| ltl_neg   : fobj_LTL -> fobj_LTL
| ltl_next  : fobj_LTL -> fobj_LTL
| ltl_until : fobj_LTL -> fobj_LTL -> fobj_LTL.
```

The interpretation of the sentences is similar to what we described in CHAPTER 5, with one major difference—the entire model can change at every time step, rather than just a designated set of state variables. As before, we define a more general auxiliary relation.

```
Fixpoint interp_LTL_aux                                              6.6.2
    (M : LTL_Mod Σ) (φ : LTL_Sen Σ) (j : nat) {struct φ} : Prop :=
  match φ with
  | ltl_true => True
  | ltl_sen ψ => M j ⊨ ψ
  | ltl_conj ψ₁ ψ₂ => interp_LTL_aux M ψ₁ j ∧ interp_LTL_aux M ψ₂ j
  | ltl_neg ψ => ¬ interp_LTL_aux M ψ j
  | ltl_next ψ => interp_LTL_aux M ψ (j + 1)
  | ltl_until ψ₁ ψ₂ =>
      ∃ k : nat, k ≥ j
        ∧ interp_LTL_aux M ψ₂ k
        ∧ (∀ i : nat, j ≤ i ∧ i < k -> interp_LTL_aux M ψ₁ i)
  end.
```

A full proof of the satisfaction condition is as follows. All cases can be discharged nearly automatically.

```
Definition LTL_sat_aux                                               6.6.3
    (σ : Σ ~> Σ') (φ : LTL_Sen Σ) (M' : LTL_Mod Σ') :
  ∀ j : nat, interp_LTL_aux M' (fmap[LTL_Sen] σ φ) j
         <-> interp_LTL_aux (fmap[LTL_Mod] σ M') φ j.
Proof.
  induction φ; cbn in *; intuition.
  all: split_hypos; exists x; intuition.
Qed.
```

The tactic split_hypos recursively searches for most valid applications of the destruct tactic and applies it.

It's hard to say how useful this institution is since the entire first-order model can change at every time step. Such anarchy is rarely desirable. It seems more useful to designate only a subset of the symbols in the signature as changeable over time, as we did in SECTION 5.2.

That said, it could be possible to study properties of LTL in general by studying $LTL(\mathcal{I})$—or if not $LTL(\mathcal{I})$, which may be somewhat of a toy example, then some richer conception of the essence of linear temporal logic. What assumptions must be made about $\mathcal{I}$ in order to prove interesting properties for $LTL(\mathcal{I})$, for example?

6.6.2 *Foundation-Independent Modal Logic*

Another interesting construction is $MDL(\mathcal{I})$, a generic modal logic over an arbitrary institution. This is even easier to define than $LTL(\mathcal{I})$. The signatures are inherited directly from $\mathcal{I}$. The sentences are as follows.

```
Inductive fobj_Modal {I : Institution} (Σ : Sig[I]) : Type :=
| modal_sen : Sen Σ -> fobj_Modal
| modal_neg : fobj_Modal -> fobj_Modal
| modal_box : fobj_Modal -> fobj_Modal.
```
6.6.4

The models are Kripke structures consisting of a type of 'worlds', an initial world, an accessibility relation $s \rightsquigarrow s'$ between worlds, and an $\mathcal{I}$-model for each world.

```
Record KripkeStructure
    {I : Institution} (Σ : Sig[I]) : Type := {
  world : Type ;
  initial_world : world ;
  transition : crelation world ;
  world_models : world -> Mod Σ ;
}.

Notation "s ⇝ s'" := (transition _ s s') (at level 80).
```
6.6.5

We define an auxiliary semantic entailment relation by asserting that $\Box\psi$ is true in world s if ψ is true in all worlds s' accessible from s.

```
Fixpoint interp_Modal_aux                                          6.6.6
    {I : Institution} {Σ : Sig[I]}
    (M : Modal_Mod Σ) (φ : Modal_Sen Σ)
    (s : world M) {struct φ} : Prop :=
  match φ with
  | modal_sen ψ => world_models M s ⊨ ψ
  | modal_neg ψ => ¬ interp_Modal_aux M ψ s
  | modal_box ψ => ∀ s', s ↛ s' -> interp_Modal_aux M ψ s'
  end.
```

The satisfaction condition is trivial, discharged entirely by automatic tactics.

```
Definition sat_modal                                               6.6.7
    {I : Institution} {Σ Σ' : Sig[I]}
    (σ : Σ ~> Σ') (φ : fobj_Modal Σ) (M' : KripkeStructure I Σ')
  : ∀ s : world M', interp_Modal_aux M' (fmap_Modal σ φ) s
                <-> interp_Modal_aux (fmap_Modal_Mod Σ' Σ σ M') φ s.
Proof.
  induction φ; split; cbn; intuition.
Qed.
```

The same questions asked of $LTL(\mathcal{F})$ can be asked here: What can we prove about $MDL(\mathcal{F})$? What assumptions must be made about $\mathcal{F}$ to guarantee good properties for $MDL(\mathcal{F})$?

While there aren't many concrete results in this chapter, it should give a flavour of what will be possible to achieve with the framework in both the near and long term. There are certainly more possibilities than those described here.

SEVEN

Conclusions and Future Work

Let's summarise the contributions of the thesis.

1. We introduce and motivate the work in CHAPTER 1 and cover most of the relevant mathematical background for the thesis in CHAPTER 2.

2. We encode the institution for first-order predicate logic with equality, *FOPEQ*, in Coq in CHAPTER 3, introducing some important concepts and proof techniques along the way, culminating in the proof of its satisfaction condition in SECTION 3.7.4.

3. We encode the institution *ACT* in CHAPTER 4—a moderately simplified form of *EVT* which we iterate upon in subsequent chapters—and prove its satisfaction condition over the course of SECTION 4.4.

4. In CHAPTER 5, we first encode three different institutions: *MacEVT* in SECTION 5.4, *LTL* in SECTION 5.5, and *LLTL* in SECTION 5.8. We then construct duplex institutions out of these building blocks, combining Event-B and LTL in different ways, finishing with the eventual duplex combination of *MacEVT* and *LLTL* in SECTION 5.8 which captures almost everything we want from such a combination. We indicate the work necessary to encode everything in full.

5. We discuss a number of institution-independent constructions in CHAPTER 6 and in each section indicate some ways to expand on those basic constructions. We cover institution-independent model

theory, institution morphisms, duplex institutions, pushouts and amalgamation, entailment systems, and a pair of foundation-independent logics.

Even though in CHAPTER 6 we already discussed at some length various constructions we encoded that extend the framework, we will now discuss some more speculative future directions.

Completing unfinished work and extending existing work

Some work remains unfinished. In the immediate future we want to prove the amalgamation property for *FOPEQ*, *ACT*, *EVT* and *MacEVT*. A systematic encoding of the work in Sannella and Tarlecki's book [ST11] would follow further in the future, extending the work in CHAPTER 6, and alongside this we could develop and explicitly encode proof techniques for institutions in Coq—perhaps even leading to proof automation for institutions.

More work on Event-B

We want to make the representation of Event-B in Coq more explicit. For example, there's no separation of guards and actions at the level of *MacEVT* sentences. This prevents certain useful syntactical transformations and makes it difficult to encode Event-B's proof rules, which must distinguish between guard and action.

If we could encode Event-B's proof rules, we should be able to prove that syntactical machine refinement tracks model-theoretic refinement with respect to the *MacEVT* semantics. Specifically I think it's most important to show that syntactical machine refinement implies model theoretic refinement—a kind of soundness proof for Event-B's most important feature.

Finally, tie-ins with Peter Rivière, Neeraj Kumar Singh and Yamine Aït-Ameur's EB4EB framework [RSA22] would also be fruitful. In fact, a collaboration with their research group has already been funded by EPSRC, in which we will investigate the relationship between our and

Rivière's approach to Event-B semantics, leading hopefully to a proof (in Coq) that *EVT* or *MacEVT* semantics can be formally connected to EB4EB.

More work on the framework's foundations

We mentioned in CHAPTER 6 that we're considering homotopy type theory as a new foundation for this work. Homotopy type theory's higher inductive types seem a natural syntactical presentation for many otherwise problematic concepts, such as pushouts (see SECTION 6.4). We could always replace all types (or most types) with setoids, but this would require a not insignificant overhaul of the work done. It's hard to judge how practical such a change would be, since the explicit support for homotopy type theory in Coq is still limited.

Coq's extraction mechanism

It's possible to extract OCaml, Haskell, or Scheme code from Coq code. For our purposes, the rough idea would be to extract certain certified semantics-preserving syntactical transformations encoded in Coq. Of course in practice the extraction does not *guarantee* anything about the correctness of the extracted code, but it's a great start.

This has not been explored in any real depth, but I suspect that there are a number of problems. Unfortunately, dependent typing gets in the way. Recall the terms defined in DEFINITION 2.4.7 and encoded in Coq in LISTING 3.6.8. Since Coq's code extraction targets are programming languages that do not have dependent type systems or ways to represent complex proof objects, the job of the extraction mechanism is partially to identify and throw away information that is not computationally relevant. But this is not always clear: take first-order terms as an example. It is not clear to Coq which parts of the term data type are computationally relevant, and hence extraction of terms and term translations includes a lot of junk.

By contrast, let's consider a perfectly legitimate alternative definition for terms in Coq.

```
Inductive Term' Σ : Type :=
| var (n : nat)
| term (F : Funcs Σ) (args : list (Term' Σ)).

Arguments var [Σ].
Arguments term [Σ].

Inductive Term_WF {Σ Γ} : Sorts Σ -> Term' Σ -> Prop :=
| wf_var n s : nth_error Γ n = Some s -> Term_WF s (var n)
| wf_term F args :
    List.Forall2 Term_WF (ar F) args
    -> Term_WF (res F) (term F args).

Definition Term Σ Γ s := { t : Term' Σ | @Term_WF Σ Γ s t }.
```

This first encodes terms as plain simply-typed data—something one could easily encode in a programming language without dependent types, like Haskell. This alone is of course insufficient because terms could be ill-formed in a number of ways—allowing for example the formation of manifest nonsense like push(pop). Thus we pair it with a well-formedness condition Term_WF which captures precisely the propositional (i.e. computationally irrelevant) content of the dependently typed definition for terms. Functions on terms will separately construct t : Term and a proof p : Term_WF t, and extraction will retain only the construction of t and erase the proof.

But this representation is not without its drawbacks, chief among them that terms are no longer well formed by construction in the same way as before. It's hard to say which approach is better suited for this project, but it could be worth exploring.

Add more institutions

Only a handful of institutions have been encoded so far. A particularly important institution to encode is *CASL* [Ast+02], since it captures much of what *FOPEQ* alone does not while remaining sufficiently gener-

ally useful. Further institution combinations, more complex than those developed in CHAPTER 5, would be fruitful to encode—*CSP-CASL* [Rog06], for example, and all the institutions defined in the process. Deciding on an economic way to encode these institutions and their relationships will constitute a significant challenge.

Much further in the future, the framework could become a fully formal basis for Till Mossakowski, Christian Maeder, and Klaus Lüttich's HETS tool [MML07] for heterogeneous specification.[14]

Endnotes

1 *Equity font*
https://mbtype.com/fonts/equity

2 *Concourse font*
https://mbtype.com/fonts/concourse

3 *Chuanren Wu's modification of Computer Modern*
https://thedrwu.com/posts/thicker-lm

4 *New Computer Modern*
https://ctan.org/pkg/newcomputermodern

5 *STIX fonts*
https://www.stixfonts.org

6 *Source Code Pro*
https://adobe-fonts.github.io/source-code-pro

7 *Comparison of CM with modern laser printing*
http://www.levien.com/type/cmr/gain.html

8 *Dedukti*
https://deducteam.github.io

9 *Twelf*
http://twelf.org/wiki/Main_Page

10 *Homotopy type theory in Coq*
https://github.com/HoTT/Coq-HoTT

11 *Coq formalisation of the theory of institutions*
https://github.com/ConorReynolds/coq-institutions

12 *Cars on a bridge in Event-B*
http://www.event-b.org/A_ch2.pdf

13 *Private Inductive Types*
https://coq.inria.fr/files/coq5-slides-bertot.pdf

14 *The Heterogeneous Tool Set*
http://hets.eu &
http://iks.cs.ovgu.de/~till/papers/hets-paper.pdf

Symbols

$\prod_{x:A} P(x)$	dependent product or function
$\sum_{x:A} P(x)$	dependent sum or pair
$p \cdot q$	transitivity of identity proofs
$p^{-1}, p^{\wedge}$	inverse of identity proof
Sig	signature category
Sen	sentence functor
Mod	model functor
$\models$	semantic entailment relation
$FOPEQ$	institution for first-order predicate logic with equality
EVT	institution for Event-B
ACT	institution of variable updates
$MacEVT$	institution for Event-B with trace semantics
LTL	institution for LTL
$LLTL$	institution for LTL with labels
Set	category of sets
Cat	category of (small) categories
$A \xrightarrow{\sigma} B$	morphism involving signature morphism σ

Coq Tactics

reflexivity

: Proves a goal of the form $a = b$ if a and b are definitionally equal.

cbn, simpl

: Simplifies the goal if possible—cbn is more common.

intros

: If the goal is of the form $\prod_{a:A} B(a)$ or $A \to B$, introduce $a : A$ to the context and replace the goal with $B(a)$ or B respectively.

rewrite H

: If H is an equality $a = b$ then this tactic will find free occurrences of a in the goal and replace them with b. Can optionally supply an arrow to determine the direction of rewriting.

f_equal

: Converts a subgoal of the form $f(x) = f(y)$ into $x = y$; that is, it encodes the fact that $x = y$ implies $f(x) = f(y)$. This is not the same as the *term* f_equal, even though they have the same name.

destruct H

: Breaks apart the hypothesis H and generates a subgoal for each constructor of the type of H (think case analysis).

induction H

: Similar to destruct but also generates an induction hypothesis if possible.

exact H

Proves the goal if the hypothesis H has the type of the goal.

apply H

Tries to match the conclusion of H with the goal and, if it can, replaces the goal with the premises of H (or proves the goal if H has no premises).

eapply H

Same as apply but creates existential variables when Coq cannot automatically instantiate variables.

unshelve eapply H

Like eapply but unshelves any shelved subgoals generated by eapply, turning them into explicit subgoals. This is the most common way that we use the eapply tactic.

refine H

Like exact but allows the term H to have 'holes', represented by underscores. Generates a subgoal for each hole which cannot be inferred from the context and 'shelves' subgoals that appear in other subgoals.

unshelve refine H

Like refine but unshelves any shelved subgoals generated by refine, turning them into explicit subgoals. This is the most common way that we use the refine tactic.

split

If the goal is a product $A \times B$, generates two subgoals, one for A and one for B.

esplit

Same as split but introduces (and shelves) existential variables rather than failing when they cannot be instantiated.

unshelve esplit

Like esplit but unshelves the existential variables shelved by esplit.

exists x

Instantiates a goal $\exists t : T.\ G$ with $x : T$, replacing the current subgoal with $G[t := x]$.

assert P

Adds a hypothesis $H : P$ to the current subgoal and adds a new subgoal to prove P before the current subgoal.

cut P

Identical to assert but places the new subgoal to prove P after the current subgoal. This tactic is used when you want to defer justifying the assumption P till after the main goal is proved.

simplify_eqs

Automatic tactic which tries to simplify goals of the form rew p in t. Handy if it works, but if it doesn't we resort to manual methods.

tac1; tac2

Run tac1, then run tac2 on all subgoals generated by tac1.

auto, congruence, easy, intuition, firstorder

Various automatic tactics.

*, -, +

Proof bullets, used to organise proofs with many subgoals.

{ … }

The opening brace focuses the first goal and the closing brace closes it. This is mainly useful in combination with tactics like assert, in which you usually write 'assert P. { * }' where * is a proof of P.

Index

Bibliography

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge: Cambridge University Press, 2010. DOI: 10.1017/CBO9781139195881.

[Agd] The Agda Development Team. *Agda 2*. URL: https://github.com/agda/agda.

[Ama+20] Gianluca Amato et al. 'Universal Algebra in UniMath'. 9th July 2020. URL: http://arxiv.org/abs/2007.04840.

[Are] The Arend Development Team. *Arend Proof Assistant*. JetBrains. URL: https://github.com/JetBrains/Arend.

[Ast+02] Egidio Astesiano et al. 'CASL: The Common Algebraic Specification Language'. In: *Theoretical Computer Science*. Current Trends in Algebraic Development Techniques 286.2 (17th Sept. 2002), pp. 153–196. ISSN: 0304-3975. DOI: 10/d8sjd3.

[Awo10] Steve Awodey. *Category Theory*. 2nd ed. USA: Oxford University Press, Inc., 2010. ISBN: 0-19-923718-2.

[Bar+22] Haniel Barbosa et al. 'Cvc5: A Versatile and Industrial-Strength SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9. DOI: 10/gspmhh.

[Bar10] Bruno Barras. 'Sets in Coq, Coq in Sets'. In: *Journal of Formalized Reasoning* 3.1 (Jan. 2010), pp. 29–48. DOI: 10.6092/issn.1972-5787/1695.

[BG79] Rod M. Burstall and Joseph A. Goguen. 'The Semantics of CLEAR, A Specification Language'. In: *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*. Berlin, Heidelberg: Springer-Verlag, 22nd Jan. 1979, pp. 292–332. ISBN: 978-3-540-10007-2. DOI: 10/d64zkm.

[BHL20] Andrej Bauer, Philipp G. Haselwarter and Peter LeFanu Lumsdaine. 'A General Definition of Dependent Type Theories'. 11th Sept. 2020. URL: http://arxiv.org/abs/2009.05539.

[Cap99] Venanzio Capretta. 'Universal Algebra in Type Theory'. In: *Theorem Proving in Higher Order Logics*. Ed. by Yves Bertot et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 131–148. ISBN: 978-3-540-48256-7. DOI: 10/fqj9bm.

[CD07] Denis Cousineau and Gilles Dowek. 'Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo'. In: *Typed Lambda Calculi and Applications*. Ed. by Simona Ronchi Della Rocca. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 102–117. ISBN: 978-3-540-73228-0. DOI: 10/b6kj3m.

[CH88] Thierry Coquand and Gérard Huet. 'The Calculus of Constructions'. In: *Information and Computation* 76.2 (1st Feb. 1988), pp. 95–120. ISSN: 0890-5401. DOI: 10/cfvb26.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN: 978-0-262-02665-9. URL: http://adam.chlipala.net/cpdt/.

[CK18] Łukasz Czajka and Cezary Kaliszyk. 'Hammer for Coq: Automation for Dependent Type Theory'. In: *Journal of Automated Reasoning* 61.1 (1st June 2018), pp. 423–453. ISSN: 1573-0670. DOI: 10/gdzwd5.

[Coq23] The Coq Development Team. *The Coq Proof Assistant*. Zenodo, 27th June 2023. DOI: 10.5281/zenodo.8161141.

[Dia22] Răzvan Diaconescu. 'Institution Theory'. In: *Internet Encyclopedia of Philosophy* (2022). ISSN: 2161-0002. URL: https://www.iep.utm.edu/insti-th/.

[Dij68] Edsger Wybe Dijkstra. 'A Constructive Approach to the Problem of Program Correctness'. In: *BIT Numerical Mathematics* 8.3 (1st Sept. 1968), pp. 174–186. ISSN: 1572-9125. DOI: 10/cs4g4s.

[DLP16] Ivaylo Dobrikov, Michael Leuschel and Daniel Plagge. 'LTL Model Checking under Fairness in ProB'. In: *Software Engineering and Formal Methods*. Ed. by Rocco De Nicola and Eva Kühn. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 204–211. ISBN: 978-3-319-41591-8. DOI: 10/fsg2.

[dMou+15] Leonardo de Moura et al. 'The Lean Theorem Prover (System Description)'. In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6. DOI: 10.1007/978-3-319-21401-6_26.

[Far17] Marie Farrell. 'Event-B in the Institutional Framework: Defining a Semantics, Modularisation Constructs and Interoperability for a Specification Language'. PhD thesis. National University of Ireland Maynooth, 2017. URL: http://mural.maynoothuniversity.ie/9911/.

[FZW15] Simon Foster, Frank Zeyda and Jim Woodcock. 'Isabelle/UTP: A Mechanised Theory Engineering Framework'. In: *Unifying Theories of Programming*. Ed. by David Naumann. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 21–41. ISBN: 978-3-319-14806-9. DOI: 10/gsqq6d.

[GB84] J. A. Goguen and R. M. Burstall. 'Introducing Institutions'. In: *Logics of Programs*. Ed. by Edmund Clarke and Dexter Kozen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1984, pp. 221–256. ISBN: 978-3-540-38775-6. DOI: 10/dtcsqb.

[GB92] Joseph A. Goguen and Rod M. Burstall. 'Institutions: Abstract Model Theory for Specification and Programming'. In: *J. ACM* 39.1 (Jan. 1992), pp. 95–146. ISSN: 0004-5411. DOI: 10/d9h9wf.

[GGP18] Emmanuel Gunther, Alejandro Gadea and Miguel Pagano. 'Formalization of Universal Algebra in Agda'. In: *Electronic Notes in Theoretical Computer Science*. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017) 338 (26th Oct. 2018), pp. 147–166. ISSN: 1571-0661. DOI: 10/gh36j7.

[Gil+19] Gaëtan Gilbert et al. 'Definitional Proof-Irrelevance without K'. In: *Proceedings of the ACM on Programming Languages* 3 (POPL 2nd Jan. 2019), 3:1–3:28. DOI: 10/gsnfvh.

[Gog91] Joseph A. Goguen. 'A Categorical Manifesto'. In: *Mathematical Structures in Computer Science* 1.01 (Mar. 1991), p. 49. ISSN: 0960-1295, 1469-8072. DOI: 10/dn7mkb.

[HA11] Thai Son Hoang and Jean-Raymond Abrial. 'Reasoning about Liveness Properties in Event-B'. In: *Formal Methods and Software Engineering*. Ed. by Shengchao Qin and Zongyan Qiu. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 456–471. ISBN: 978-3-642-24559-6. DOI: 10/dnsd88.

[HJ98] Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[Hoa+16] Thai Son Hoang et al. 'Foundations for Using Linear Temporal Logic in Event-B Refinement'. In: *Formal Aspects of Computing* 28.6 (1st Nov. 2016), pp. 909–935. ISSN: 0934-5043. DOI: 10/f864wr.

[Hod01] Wilfrid Hodges. *Logic*. 2nd ed. Penguin Books, 2001. ISBN: 978-0-14-100314-6.

[Hod93] Wilfrid Hodges. *Model Theory*. Cambridge University Press, 1993.

[HS98] Martin Hofmann and Thomas Streicher. 'The Groupoid Interpretation of Type Theory'. In: *Twenty Five Years of Constructive Type Theory*. Ed. by Giovanni Sambin and Jan M Smith. Oxford University Press, 15th Oct. 1998. ISBN: 978-0-19-850127-5. DOI: 10.1093/oso/9780198501275.003.0008.

[Jon74] K. Jon Barwise. 'Axioms for Abstract Model Theory'. In: *Annals of Mathematical Logic* 7.2 (1st Dec. 1974), pp. 221–265. ISSN: 0003-4843. DOI: 10/bx4kfn.

[Kna+15] Alexander Knapp et al. 'An Institution for Simple UML State Machines'. In: *Fundamental Approaches to Software Engineering*. Ed. by Alexander Egyed and Ina Schaefer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 3–18. ISBN: 978-3-662-46675-9. DOI: 10/gjsjxn.

[Lam77] L. Lamport. 'Proving the Correctness of Multiprocess Programs'. In: *IEEE Transactions on Software Engineering* SE-3.2 (Mar. 1977), pp. 125–143. ISSN: 1939-3520. DOI: 10.1109/TSE.1977.229904.

[Lam99] Leslie Lamport. 'Specifying Concurrent Systems with TLA+'. In: *Calculational System Design* (Apr. 1999), pp. 183–247. URL: https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/.

[Ler09] Xavier Leroy. 'Formal Verification of a Realistic Compiler'. In: *Communications of the ACM* 52.7 (1st July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10/c9sb7q.

[LS19] Andreas Lynge and Bas Spitters. 'Universal Algebra in HoTT'. In: TYPES. 2019. URL: http://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_7.

[Mes89] José Meseguer. 'General Logics'. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by H. -D. Ebbinghaus et al. Vol. 129. Logic Colloquium'87. Elsevier, 1989, pp. 275–329. DOI: 10/brw9bq.

[MML07] Till Mossakowski, Christian Maeder and Klaus Lüttich. 'The Heterogeneous Tool Set, Hets'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Orna Grumberg and Michael Huth. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 519–522. ISBN: 978-3-540-71209-1. DOI: 10/b9w795.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Cham: Springer International Publishing, 2014. ISBN: 978-3-319-10541-3. DOI: 10/gjmn46.

[nLa23] nLab authors. *Pushout Type*. Aug. 2023. URL: https://ncatlab.org/nlab/show/pushout+type.

[OKe04] Greg O'Keefe. 'Towards a Readable Formalisation of Category Theory'. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of Computing: The Australasian Theory Symposium (CATS) 2004 91 (16th Feb. 2004), pp. 212–228. ISSN: 1571-0661. DOI: 10/cvnsjp.

[Pie+23a] Benjamin C. Pierce et al. *Logical Foundations*. Ed. by Benjamin C. Pierce. Vol. 1. Software Foundations. Electronic textbook, 2023.

[Pie+23b] Benjamin C. Pierce et al. *Programming Language Foundations*. Ed. by Benjamin C. Pierce. Vol. 2. Software Foundations. Electronic textbook, 2023.

[PL10] Daniel Plagge and Michael Leuschel. 'Seven at One Stroke: LTL Model Checking for High-Level Specifications in B, Z, CSP, and More'. In: *International Journal on Software Tools for Technology Transfer* 12.1 (1st Feb. 2010), pp. 9–21. ISSN: 1433-2787. DOI: 10/bj2svh.

[Pnu77] Amir Pnueli. 'The Temporal Logic of Programs'. In: *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*. 18th Annual Symposium on Foundations of Computer Science (Sfcs 1977). Oct. 1977, pp. 46–57. DOI: 10/dn8cpn.

[Rey21] Conor Reynolds. 'Formalizing the Institution for Event-B in the Coq Proof Assistant'. In: *Rigorous State-Based Methods*. Ed. by Alexander Raschke and Dominique Méry. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 162–166. ISBN: 978-3-030-77543-8. DOI: 10/gpjt7c.

[Rie17] E. Riehl. *Category Theory in Context*. Aurora: Dover Modern Math Originals. Dover Publications, 2017. ISBN: 978-0-486-82080-4.

[RK13] Florian Rabe and Michael Kohlhase. 'A Scalable Module System'. In: *Information and Computation* 230 (1st Sept. 2013), pp. 1–54. ISSN: 0890-5401. DOI: 10.1016/j.ic.2013.06.001.

[RM22] Conor Reynolds and Rosemary Monahan. 'Machine-Assisted Proofs for Institutions in Coq'. In: *Theoretical Aspects of Software Engineering*. Ed. by Yamine Aït-Ameur and Florin Crăciun. Cham: Springer International Publishing, 2022, pp. 180–196. ISBN: 978-3-031-10363-6. DOI: 10/gqfnb6.

[RM24] Conor Reynolds and Rosemary Monahan. 'Reasoning about Logical Systems in the Coq Proof Assistant'. In: *Science of Computer Programming* 233 (1st Mar. 2024), p. 103054. ISSN: 0167-6423. DOI: 10/gs7f2x.

[Rog+22] Markus Roggenbach et al. *Formal Methods for Software Engineering: Languages, Methods, Application Domains*. Texts in Theoretical Computer Science. An EATCS Series. Cham: Springer International Publishing, 2022. ISBN: 978-3-030-38799-0 978-3-030-38800-3. DOI: 10.1007/978-3-030-38800-3.

[Rog06] Markus Roggenbach. 'CSP-CASL—A New Integration of Process Algebra and Algebraic Specification'. In: *Theoretical Computer Science*. Algebraic Methods in Language Processing 354.1 (21st Mar. 2006), pp. 42–71. ISSN: 0304-3975. DOI: 10/cwjgw4.

[Ros+21] Tobias Rosenberger et al. 'Institution-Based Encoding and Verification of Simple UML State Machines in CASL/SPASS'. In: *Recent Trends in Algebraic Development Techniques*. Ed. by Markus Roggenbach. Cham: Springer International Publishing, 2021, pp. 120–141. ISBN: 978-3-030-73785-6. DOI: 10.1007/978-3-030-73785-6_7.

[Roş18] Grigore Roşu. 'Finite-Trace Linear Temporal Logic: Coinductive Completeness'. In: *Formal Methods in System Design* 53.1 (1st Aug. 2018), pp. 138–163. ISSN: 1572-8102. DOI: 10.1007/s10703-018-0321-3.

[RSA22] Peter Rivière, Neeraj Kumar Singh and Yamine Aït-Ameur. 'EB4EB: A Framework for Reflexive Event-B'. In: *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS). Mar. 2022, pp. 71–80. DOI: 10/gq98f3.

[RT13] Alexander Romanovsky and Martyn Thomas, eds. *Industrial Deployment of System Engineering Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-33169-5 978-3-642-33170-1. DOI: 10/jn5k.

[RV02] Alexandre Riazanov and Andrei Voronkov. 'The Design and Implementation of VAMPIRE'. In: *AI Communications* 15.2,3 (1st Aug. 2002), pp. 91–110. ISSN: 0921-7126.

[Sch+14] Steve Schneider et al. 'Managing LTL Properties in Event-B Refinement'. In: *Integrated Formal Methods*. Ed. by Elvira Albert and Emil Sekerinski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 221–237. ISBN: 978-3-319-10181-1. DOI: 10/gsnfvj.

[Shu13] Mike Shulman. *From Set Theory to Type Theory*. The n-Category Café. 7th Jan. 2013. URL: https://golem.ph.utexas.edu/category/2013/01/from_set_theory_to_type_theory.html.

[Soz+20] Matthieu Sozeau et al. 'The MetaCoq Project'. In: *Journal of Automated Reasoning* 64.5 (1st June 2020), pp. 947–999. ISSN: 1573-0670. DOI: 10.1007/s10817-019-09540-0.

[Soz10] Matthieu Sozeau. 'Equations: A Dependent Pattern-Matching Compiler'. In: *Interactive Theorem Proving*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 419–434. ISBN: 978-3-642-14052-5. DOI: 10/b6ncs7.

[ST11] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. Springer Berlin, Heidelberg, 2011. 584 pp. ISBN: 978-3-642-17335-6. DOI: 10.1007/978-3-642-17336-3.

[UFP14] UFP (The Univalent Foundations Program). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2014. URL: https://homotopytypetheory.org/book.

[Wie14] John Wiegley. *Jwiegley/Category-Theory*. 2014. URL: https://github.com/jwiegley/category-theory.