Proof Synthesis with Free Extensions in Intensional Type Theory

Nathan Corbyn
King's College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the Computer Science Tripos, Part III

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: `nc513@cam.ac.uk`

June 12, 2021

Declaration

Signed:

Date:

Acknowledgements

I would like to thank my supervisor Jeremy Yallop for suggesting this project and encouraging me throughout. I would also like to thank Ohad Kammar and Guillaume Allais for some very useful discussions regarding my formalisation of universal algebra. Finally, I'd like to thank Timothy Griffin for the enormous amount of support he's offered me over the past four years and for inspiring my move into theory.

Proof Synthesis with Free Extensions in Intensional Type Theory

Abstract

Recent developments in the foundations of mathematics have led to a surge of interest in intensional theories of types and their applications in verified programming & formalised mathematics. Due to their constructive nature, these theories generally cannot benefit from classical proof automation techniques, but concurrently require a great deal of 'bookkeeping' to work with their proof-relevant notions of identity. With the aim of eliminating some of this burden, this dissertation discusses the application of a class of mathematical constructions known as free extensions to the problem of proof synthesis in intensional type theory. Specifically, this work describes the design and implementation of an extensible tactic for the Agda proof assistant, capable of synthesising proofs of algebraic identities. This tactic is formally verified as sound and complete, does not rely on postulates or extensionality, and is compatible with a broad variety of Agda configurations.

11797 words

Contents

1 — Introduction

This dissertation discusses the application of a class of mathematical constructions known as free extensions to the problem of proof synthesis in intensional type theory. Specifically, this work describes the design and implementation of an extensible tactic for the Agda proof assistant[17], capable of synthesising proofs of algebraic identities. This tactic is formally verified as sound and complete, does not rely on postulates or extensionality, and is compatible with a broad variety of Agda configurations.

This chapter gives a brief overview of the problem space and describes the primary contributions of this work. Chapter 2 builds on this discussion, motivating the problem and touching briefly on existing approaches. Chapter 3 then provides a detailed introduction to free extensions, generalising their definition to a broad class of equational systems described by Fiore & Hur[8]. Following this, chapters 4 and 5 describe my formalisation of these concepts in the Agda proof assistant and how they can be integrated into a tactic using Agda's proof reflection tools. Finally, chapter 6 concludes the dissertation, summarising my findings.

1.1 Overview

It is well-known that dependent type theories are highly amenable for use both as foundations for constructive mathematics and for verified programming[14]. While many dependently typed programming languages based on these type theories exist[4, 17, 18], we are yet to see their widespread application outside of academia.

From personal experience, I believe the primary barrier to the widespread deployment of these systems is an obstructively tedious user experience. Peter Hancock famously said of the Coq proof assistant[18] "Using Coq is like doing brain surgery over the telephone". The lack of automation in Agda[19] and slow type-checking performance makes it similarly difficult to use for large-scale formalisation and verification projects.

Compounding these issues is the problem of extensionality. It is generally desirable for dependent type systems to build on type theories with decidable type-checking. However, supporting decidable type-checking requires we make a theoretical distinction between a theory's internal notion of identity and its equality judgements[12]. Type theories with this distinction are said to be *intensional* (as opposed to *extensional*). Regrettably, many popular intensional type theories cannot directly support key extensional concepts found throughout mathematics and computer science (e.g., subobjects and quotients). While there are many proposed embeddings of these constructions in such theories, they remain largely unsupported in major implementations. Instead, users are required to work under these embeddings explicitly, potentially postulating additional axioms. For example, it is common to work with so-called *setoids* (sets equipped with an equivalence relation) in place of sets in order to support quotients[3], or to postulate function extensionality such that operationally identical functions can be identified. Taken together, these issues adversely affect the readability of proofs and the canonicity properties that type theories usually enjoy, further hurting the user experience.

Rather than adding to the body of work proposing novel type theories, better suited to the formalisation of extensional constructs (e.g., homotopy type theory[16]), this work leverages existing techniques in order to simplify the task of developing proofs in intensional type theories as they exist in mainstream implementations today. In particular, recent work in the field of multi-stage programming, due to Yallop *et al.* highlights the applicability of a class of algebraic constructions, known as *free extensions*, to the problem of optimising staged computations using algebraic identities[20]. Further work from Allais *et al.* demonstrates that these constructions have additional applications in dependent type systems, efficiently dealing with proof obligations in terms indexed by algebraic computations[1]. This dissertation extends this work to the problem of synthesising proofs of algebraic identities in Martin-Löf's intensional type theory[15], as implemented in the Agda proof assistant, for arbitrary finitary, mono-sorted equational theories.

1.2 Contributions

The primary contribution of this work is an extensible tactic for the Agda proof assistant, capable of synthesising proofs of algebraic identities. This tactic is parametric in the algebraic solver (free extension) that it uses and can therefore be extended to support any of a broad class of equational theories. The tactic is packaged alongside a universe-polymorphic formalisation of finitary, mono-sorted universal algebra and equational logic, designed to be compatible with the Agda standard library's existing characterisations of algebraic structures. This formalisation enables, for the first time, the specification and verification of free extensions for equational theories in terms of a universal property, thus guaranteeing the tactic is sound and complete for any theory. To demonstrate the capability of this framework, this work further contributes verified solvers for semigroups and commutative semigroups (defined over arbitrary setoids) alongside an extensive collection of example proofs.

2 — Background

This chapter introduces various background material in order to motivate the problem of proof synthesis. §2.1 outlines key concepts from dependent type theory, assuming an understanding of simple types. §2.2 takes these concepts into the context of an implementation (Agda), demonstrating typical frustrations of proof development. Finally, §2.3 concludes with a summary, stating the aims of this project.

2.1 Dependent Types

This section outlines the basic concepts in dependent type theory as relevant to motivating this work; for a more exhaustive introduction, see [16]. This outline is limited to Martin-Löf's intensional type theory (MLTT)[15], and adopts similar notation to that used by the Univalent Foundations Program in [16].

2.1.1 Motivation

Simple type theories guarantee various classes of undesirable programs are ill-typed (e.g,. meaningless applications, such as $3 + \textbf{true}$), but cannot reason about program correctness in full generality. Conventionally, proving correctness properties is done by developing a semantics for the language in question, and appealing to this when formalising correctness arguments.

While fundamentally correct, this approach is unsatisfactory for several reasons. First, we are required to formalise our semantics and develop its meta-theory. This is an enormous challenge, even for relatively simple languages. Second, a program and its proofs live in two distinct environments: the development environment and a separate proof assistant, adding to the burden of maintenance. This separation of program from proof is particularly frustrating when working in a typed λ-calculus. The Curry-Howard correspondence tells us that the type system of such a calculus has logical content. It therefore seems reasonable to expect to be able to state and prove correctness properties using this logic.

Unfortunately, simple type theories suffer from fundamental limitations in their expressive power, meaning that this kind of reasoning is not possible. In particular, the logical content of a type theory comes from our interpretation of propositions as types. While program correctness properties are normally stated and proven using higher-order logic, non-dependent type theories are limited in their presentations of quantification and identity, preventing them from capturing predicate calculus in full generality. Bridging this gap is the primary motivation for the development of dependent type theories.

2.1.2 Quantification

Fundamental to the notion of quantification is the idea of a predicate. In classical set theory, a predicate can be described as a truth-valued function. Martin-Löf makes the observation that under propositions-as-types, a predicate therefore corresponds to a 'type-valued' function. This idea is made precise in terms of indexed families of types.

Suppose we have a type A and, for every term $x : A$, a type $B(x)$. Under propositions-as-types, constructing an element of $B(x)$ given some specific choice of x corresponds to proving one of an A-indexed family of propositions – or, equivalently, proving the predicate B holds at the point x.

From this understanding of predicates in type theory, the Martin-Löf interpretation of quantifiers follows naturally. For example, given some A-indexed type family B, what does it mean to prove *'for all $x : A$, $B(x)$ '*? It is easy to see that constructing an element of $B(x)$ given an arbitrary $x : A$ amounts to writing down some function taking each $x : A$ to an element of $B(x)$. However, a function f whose return type *depends on* its input is untypable using simple types. To rectify this, Martin-Löf introduces the type-former Π (*dependent product*), with the following introduction and elimination rules (assuming implicitly that all contexts and types are well-formed):

$$\frac{\Gamma, x : A \vdash t : B(x)}{\Gamma \vdash \lambda(x : A).t : \prod_{x:A} B(x)} \ (\Pi\mathrm{I})$$

$$\frac{\Gamma \vdash f : \prod_{x:A} B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash f\,a : B(a)} \ (\Pi\mathrm{E})$$

The introduction gives us the typing $f : \prod_{x:A} B(x)$, usually read *'f is a dependent function from A to B'*, or *'f is a proof that for all $x : A$, $B(x)$'*, while the elimination rule corresponds to application of f, or logically speaking, universal instantiation. Note that these rules correspond directly to the usual introduction and elimination rules for $\forall$.

Similarly, we can ask what it means to prove '*there exists* an $x : A$ such that $B(x)$'. In an intuitionistic interpretation of existence, proving this corresponds to exhibiting some witness $a : A$, alongside a proof of $B(a)$. Simply put, we are interested in pairs $(a : A, b : B(a))$. As any such pair proves our formula, an encoding of this formula as a type reduces to pairs with a second component whose type *depends on* the value of its first component. As with $\forall$, this encoding of existence cannot be captured using simple types, leading Martin-Löf to introduce the type-former Σ (*dependent sum*),

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B(a)}{\Gamma \vdash (a,b) : \sum_{x:A} B(x)}\ (\Sigma\mathrm{I})$$

$$\frac{\Gamma \vdash p : \Sigma_{x:A} B(x) \qquad \Gamma, x : A, y : B(a) \vdash g : C(x,y)}{\Gamma \vdash \mathrm{ind}_{\sum_{x:A} B(x)}(C, x, y.g, p) : C(p)}\ (\Sigma\mathrm{E})$$

where the introduction rule corresponds directly to the construction of a witness of B. In the elimination rule, $\mathrm{ind}_{\sum_{x:A} B(x)}$ is the *induction principal* for dependent sums. Informally, this states that to prove the predicate C holds for an arbitrary dependent pair, it is enough to show that it holds for all *canonical* pairs (i.e., pairs of the form (a,b) for some $a : A$ and $b : B(a)$). Note again that these rules correspond directly to the introduction and elimination rules for $\exists$ (in the intuitionistic sense).

2.1.3 Identity

One subtle issue of dependent type theory is the problem of identity propositions, i.e., sentences of the form 'x and y are identical' which we intend to encode as types.

Identity propositions are encoded as types using the type-former Id. Given any type A and $x, y : A$, $Id_A(x,y)$ is understood as the type of proofs that x and y are identical elements of A. Id comes equipped with the following introduction and elimination rules:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{refl}_A a : Id_A(a,a)}\ (Id\mathrm{I})$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : Id_A(a,b) \qquad \Gamma, x : A \vdash c : C(x, x, \mathrm{refl}_A x)}{\Gamma \vdash \mathrm{ind}_{Id_A}(C, x.c, a, b, p) : C(a,b,p)}\ (Id\mathrm{E})$$

where the introduction rule corresponds to proof by reflexivity. In the elimination rule, ind_{Id_A} is the induction principal for identities and states that to prove some predicate C holds for an identification $p : Id_A(a,b)$, it is enough to show that C holds for reflexivity at every $x : A$.

2.1.4 Judgemental Equality & Extensionality

Using the notion of identity introduced in §2.1.3, we might hope to be able to prove simple facts about arithmetic. For example, assuming standard definitions for $+$ and $\times$, we might expect the following typing to be derivable:

$$\lambda(x : \mathbb{N}).\mathrm{refl}_{\mathbb{N}}(2 \times x) : \prod_{x:\mathbb{N}} Id_{\mathbb{N}}(2 \times x, x + x) \tag{2.1}$$

However, applying the typing rules introduced so far in this chapter, we find this term actually types as follows:

$$\cfrac{\cfrac{\cfrac{\cdots}{x : \mathbb{N} \vdash 2 \times x : \mathbb{N}}}{x : \mathbb{N} \vdash \mathrm{refl}_{\mathbb{N}}(2 \times x) : \mathit{Id}_{\mathbb{N}}(2 \times x, 2 \times x)}\ (\mathit{Id}\mathrm{I})}{\cdot \vdash \lambda(x : \mathbb{N}).\mathrm{refl}_{\mathbb{N}}(2 \times x) : \prod_{x:\mathbb{N}} \mathit{Id}_{\mathbb{N}}(2 \times x, 2 \times x)}\ (\Pi\mathrm{I})$$

It is reasonable to expect that, as $2 \times x$ will β-reduce to $x + x$, we can simply replace $\mathit{Id}_{\mathbb{N}}(2\times x, 2\times x)$ with $\mathit{Id}_{\mathbb{N}}(2\times x, x+x)$. However, the deduction system described so far has no notion of β-conversion, making this kind of substitution impossible without use of informal reasoning. Formalising this idea motivates the introduction of a notion of *judgemental* (sometimes *definitional*) *equality*.

Judgemental equalities take one of two forms: $\Gamma \vdash a \equiv b : A$, stating that in context Γ, a and b are equal terms of type A; and $\Gamma \vdash A \equiv B$, stating that in context Γ, A and B are equal types. These two judgements are defined mutually inductively, but can be identified in the presence of universe types (§2.1.5).

The inductive definition of judgemental equality contains $\beta\eta$-conversions for all type constructors, alongside rules ensuring that both judgements induce equivalence relations on terms and types in a given context. Taken with the following rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash a : B}$$

computation inside of types becomes possible, making 2.1 derivable.

Critically, judgemental equality is distinct from identity as discussed in §2.1.3: the first is as a purely meta-theoretic notion, while the second is internal to the theory. Judgemental equalities can be used in order to derive typings involving some computation, but are not first class objects, preventing proof terms from referring to them.

With a notion of judgemental equality we can also define *canonicity*. A type theory is $\mathbb{N}$-*canonical*, if every closed term of type $\mathbb{N}$ is judgementally equal to a numeral. This implies that every term of every type is $\beta\eta$-equivalent to a 'canonical form' (e.g., every dependent pair is definitionally equal to a pair (a, b)), as given a non-canonical element of $\mathbb{N}$, the induction principle for $\mathbb{N}$ allows for the definition of a non-canonical element of any inhabited type. Proving canonicity is equivalent to proving that the computational interpretation of our type theory forms a confluent strongly-normalising rewrite system. Under the Curry-Howard correspondence, this implies the consistency of the logical content of our type theory (relative to the meta-theory in which canonicity was proven).

Identity types identify strictly more terms than equality judgements; as while the above remarks guarantee the following rule is admissible,

$$\frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash \mathrm{refl}_A a : Id_A(a,b)}$$

the reciprocal is not the case. For example, it is elementary to give a term of type

$$\prod_{x,y,z:\mathbb{N}} Id_{\mathbb{N}}((x+y)+z, x+(y+z))$$

i.e., it is trivial to prove

$$\forall x,y,z \in \mathbb{N}.\, (x+y)+z = x+(y+z)$$

However, we cannot derive the judgement

$$x:\mathbb{N}, y:\mathbb{N}, z:\mathbb{N} \vdash (x+y)+z \equiv x+(y+z) : \mathbb{N}$$

as neither the left-hand side nor the right-hand side contain β-reducible sub-terms. In other words, the left and right-hand sides differ *intensionally*, as unequal pieces of syntax, but have provably identical *extensions*.

We could bridge this gap by identifying these two notions of equality. For this purpose, the following rule is sufficient:

$$\frac{\Gamma \vdash p : Id_A(a,b)}{\Gamma \vdash a \equiv b : A}$$

However, this would have the undesirable effect of making type-checking undecidable, as deriving a judgemental equality could now involve synthesising a proof of an arbitrary theorem. It is therefore common to omit this rule, leaving us with an *intensional* type theory (as opposed to an *extensional* theory). Instead, when extensionality is required, axioms such as *uniqueness of identity proofs* and K can be introduced, postulating that all identity proofs are judgementally equal to proofs by reflexivity. The introduction of either of these axioms to an intensional type theory yields a *propositionally extensional* type theory, but does not impact the decidability of type-checking.

However, there are many reasons for the interest in intensional type theories beyond decidablility of type-checking. In particular, omitting extensionality allows for a richer class of models, some of which are of particular theoretical interest. For example, recent work from the Univalent Foundations Program introduces a homotopical interpretation of identity types[16]. This interpretation of identity types is not possible in an extensional theory, and has been shown to be inconsistent with the two axioms mentioned above.

To summarise, intensional MLTT is fundamentally weaker than its extensional variants, but is of great interest due to its desirable type-checking properties and the non-standard interpretations of identity that it supports. By focusing on pure intensional MLTT, this work remains applicable to any of the many extensions to MLTT that have been developed, including its extensional variants.

2.1.5 Universes

A final idea from dependent type theory of particular significance to this work is the idea of *universe types*. Up until this point, our use of quantification has been restricted to quantifying over elements of some type, but not types themselves (i.e., the first-order case). Universe types allow us to support higher-order quantification with minimal adjustment to our deductive system.

A universe $\mathcal{U}$ is a type whose elements are themselves types. We further require that universes are closed under type formation. For example, given $A, B : \mathcal{U}$, we can form the type $A \to B$, which, by closure, should too be an element of $\mathcal{U}$.

Indexed families of types have a natural encoding using universes. In particular, every A-indexed type family whose codomain lives in a universe $\mathcal{U}$ corresponds to an element of $A \to \mathcal{U}$. Many authors therefore identify the two, defining dependent sums and products in terms of functions into universe types.

In its original presentation, MLTT assumed a single universe $\mathcal{U}$, of which all types were elements, including $\mathcal{U}$ itself. While this seems like an attractive idea, unfortunately, taking $\mathcal{U} : \mathcal{U}$ leads to a contradiction by way of Girard's paradox (a type-theoretic analogue of Russell's paradox). To avoid such paradoxes, contemporary presentations instead assume a predicative hierarchy of universe types,

$$\mathcal{U}_0 : \mathcal{U}_1 : \ldots : \mathcal{U}_n : \ldots$$

with the property that a type A is well-formed in a context Γ if and only if there exists an i such that $\Gamma \vdash A : \mathcal{U}_i$. It is also common to assume *universe cumulativity* (i.e., if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$). However, by omitting this assumption, we retain the desirable property that the theory has unique typing (up to $\beta\eta$-equivalence).

2.2 Dependent Types in Practice

With the core concepts of dependent type theory outlined, this section discusses their presentation in the context of an implementation. Specifically, this section introduces several simple proofs, as formalised in the Agda proof assistant. The principal aim here is to point out typical frustrations associated with proof development in Agda, but this section also provides a basis for comprehension of the more complex proofs presented in later chapters.

2.2.1 A First Example

The notation Agda uses differs superficially from that of §2.1. For example, Π-types are introduced using ∀, and identity types using _≡_. Leveraging type inference algorithms, Agda also allows arguments to be marked as *hidden* (using {–}), suggesting to the type-checker that, wherever possible, these arguments should be inferred automatically. While various other notational differences exist, these will be explained as necessary.

As a first example, consider the following proof of a trivial theorem of arithmetic,

```
simple : ∀ x y → x + (2 + (3 + y)) ≡ x + (5 + y)
simple x y = refl
```

taking note of the differences in notation mentioned above. In this case, inference of hidden arguments allows the constructor refl to be used without any additional information.

2.2.2 Variations on a Theme

Building on this example, consider the following proof of an equivalent theorem,

```
variation₁ : ∀ x y → (x + 2) + (3 + y) ≡ x + (5 + y)
variation₁ x y = +-assoc x 2 (3 + y)
```

Notice that in place of refl, this proof makes use of the associativity of _+_, a result proven in Agda's standard library. This choice of proof term is not arbitrary: refl cannot be applied here.

To those unfamiliar with interactive proof assistants, this may seem absurd. The two theorems differ only in the placement of their brackets, and associativity tells us that this is inconsequential. When commutativity is involved, too, the proof becomes drawn out,

```
variation₂ : ∀ x y → (2 + x) + (y + 3) ≡ x + (y + 5)
variation₂ x y = begin
    (2 + x) + (y + 3)
```

```
≡⟨ cong₂ _+_ (+-comm 2 x) (+-comm y 3) ⟩
  (x + 2) + (3 + y)
≡⟨ +-assoc x 2 (3 + y) ⟩
  x + (5 + y)
≡⟨ cong₂ _+_ refl (+-comm 5 y) ⟩
  x + (y + 5)
∎
```

Given the apparent simplicity of these examples, it is entirely reasonable to question why the type-checker would refuse to accept proofs by reflexivity. However, recall that proofs by reflexivity are only permitted in situations where we can demonstrate the identity holds judgementally. The fact that these results are a consequence of derived algebraic properties of _+_ (i.e., they do not follow from oriented $\beta\eta$-reduction) means that they cannot hold judgementally, and, as such, reflexivity is not typeable in this context. This is a consequence of the theoretical distinction between identity types and equality judgements discussed in §2.1.4.

Of course, substituting x and y for any numerals m and n will result in an identity that holds judgementally by way of a series of simple β-reductions. The problem here stems from the fact that x and y are not numerals, but rather bound variables. The definition of _+_ is given by cases on its first argument, meaning that in situations where its first argument is not a numeral (i.e., non-canonical), it is impossible to determine how an application should be reduced.

In general, non-canonical terms that cannot be reduced due to the presence of one or more free variables are said to be *open*. As demonstrated by the examples above, open sub-terms greatly diminish the type-checker's ability to derive judgemental equalities, generally requiring it to fall back on a naïve syntax-directed approach. Such an approach will fail in all but the most trivial cases, hence the requirement for explicit proofs.

2.2.3 Spelling Out the Problem

The problem identified in §2.2.2 is not that our notion of judgemental equality is too weak. Strengthening equality judgements to capture examples such as those above would come at the cost of intensionality, which, as discussed in §2.1.4, is something that we would like to preserve. Rather, the problem is the fact that human effort is required to derive identity proofs in situations where it is entirely possible to do so mechanically. In other words, we want to be able to synthesise identity proofs automatically, just knowing the algebraic properties of the operators involved.

2.2.4 Algebraic Solvers & Reflection

The problem identified in §2.2.3 is well known, and has been approached with a variety of techniques in a number of settings. In proof assistants with a dedicated language for authoring tactics (e.g., Coq), one can imagine writing a proof search algorithm that uses simple algebraic simplifications in order to synthesise proofs of the kind seen so far in this chapter.

Without such a language, Agda relies on internal algebraic solvers, such as the Agda standard library's ring solver, for this task. For example, the ring solver can be applied to the previous example as follows:

```
import Algebra.Solver.Ring.NaturalCoefficients.Default as Solver

automated : ∀ x y → (2 + x) + (y + 3) ≡ x + (y + 5)
automated = solve 2 (λ x y → (con 2 :+ x) :+ (y :+ con 3) :=
                                    x :+ (y :+ con 5))
                            refl
  where open Solver +-*-commutativeSemiring
```

Here, solve accepts as arguments the number of free variables in its goal; a description of the abstract syntax of its goal; and a proof that the normal forms it computes for each side are identical. While verbose, this is considerably simpler than a handwritten proof.

Notice that, assuming the normalised identity always holds judgementally, all of the information passed to solve is statically available. In early versions of Agda, this information was inaccessible to Agda code. However, since the addition of Agda's reflection system, it has been possible to write code capable of inferring this information automatically and generating an appropriate call to solve. Exploiting this, the above proof simplifies further to,

```
reflection : ∀ x y → (2 + x) + (y + 3) ≡ x + (y + 5)
reflection = solve-∀
  where open import Data.Nat.Tactic.RingSolver
```

2.2.5 The Catch

Solvers capable of synthesising proofs of the identities we are interested in exist for a variety of algebraic structures. Implementations of a number of these can be found in Agda's standard library. A handful of these implementations have been enhanced with proof reflection similar to that shown above. There is, however, a catch.

Development of these solvers is fragmented, with most specified, implemented and verified independently. The more complex examples, such as the ring solver, are split across dozens of modules, and consist of thousands of lines of code, but provide little opportunity for reuse. Furthermore, enhancements based on proof reflection are tailor-made for each solver and are themselves complex programs which cannot be verified due to the limitations of Agda's reflection system.

Fundamentally, each of these solvers performs the same task: deciding equivalence of open terms in some equational theory. This suggests some level of redundancy in current approaches to their design and implementation, as their specifications are identical, modulo the theory they decide. This approach not only results in wasted effort, but also limits the ways in which algebraic solvers can be composed. Similar points can be made for the application of reflection.

The objective of this project is to tackle the problem of proof synthesis with a single unified approach. The ultimate goal of the project is to develop a maximally reusable framework for the specification, implementation and verification of algebraic solvers, enhanced with proof reflection.

2.3 Summary

This chapter outlined the key concepts that distinguish dependent type theories from their simple counterparts. Taking these concepts into the context of Agda'a implementation, we saw that even trivial results can involve lengthy proofs due to the limited strength of equality judgements. However, as discussed, a stronger equality judgement comes at the cost of losing the desirable property of intensionality.

As seen in §2.2, algebraic solvers can be applied to alleviate these burdens with great effect, particularly when enhanced using proof reflection (§2.2.4). That being said, their development has so far been scattered, resulting not only in redundancy, but also limited composability. This project aims to provide a unified solution to the problem of proof synthesis, while maintaining applicability to a broad range of type theories based on MLTT.

2.3.1 Project Aims

Specifically, this project aims to:

1. Develop a single framework in which arbitrary algebraic solvers can be specified and verified.
2. Ensure the framework retains compatibility with a broad family of type theories based on intensional MLTT.
3. Demonstrate the capability of this framework by implementing and verifying a collection of algebraic solvers.
4. Enhance this framework with a single tactic, parametric in its algebraic solver, eliminating the need for tailor-made solutions for different classes of algebra.
5. Produce a collection of example proofs, synthesised using the tactic.

3 — Free Extensions

The previous chapter demonstrated the opportunity for a unified approach to the development of verified algebraic solvers. However, it gave no suggestion as to the nature of such an approach. To this end, this chapter describes a class of mathematical constructions known as free extensions, demonstrating their applicability in this context.

Free extensions are a well-understood concept in universal algebra, but are rarely discussed in the literature. In fact, this chapter argues that, while unacknowledged, free extensions already serve as the foundation for a variety of real-world proof-synthesis solutions. From this, it follows that the problem of providing a general-purpose framework for the development of verified algebraic solvers reduces to the problem of specifying free extensions for arbitrary equational systems.

Overview §3.1 discusses the problem of discovering algebraic identities in the context of abstract syntax. This discussion leads to an informal notion of free extension, which the remainder of this chapter attempts to capture formally. In particular, §3.2 recalls the definition of a broad class of equational systems described by Fiore & Hur[8]. This serves as a foundation for §3.3, which provides a general definition of free extensions in this setting and discusses their relationship with normalisation-by-evaluation techniques. §3.4 then concludes with a brief summary.

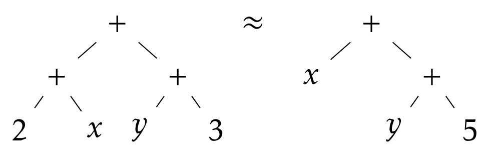
3.1 Abstract Syntax & Normal Forms

Let x and y be arbitrary natural numbers. It goes without saying that

$$(2+x)+(y+3) = x+(y+5) \tag{3.1}$$

However, this judgement is based on our intuitions about +, and its various algebraic properties. A proof assistant has a very different view.

While we, as humans, see the equation above and immediately infer its semantic content based on our experience with arithmetic, a proof assistant will treat the left-hand and right-hand sides of this equation as syntactic objects. Thus, from the perspective of a proof assistant, 3.1 is better pictured as



where $\approx$ (informally) encodes our notion semantic equivalence, as applied to syntax trees. It should now be clear why the validity of this equation is not immediately obvious to a proof assistant: the two sides are not equal *as trees*.

As discussed in the previous chapter, in situations where the values of x and y are known, we can apply a series of β-reductions in order to derive this equivalence. In this situation, this is not the case. Instead, we have a pair of *partially static* expressions: each contains some *dynamic* sub-terms (e.g., x and y), the values of which are as yet undetermined; and some *static* sub-terms (e.g., $2, 3$ and 5), the values of which are known. In proving this equivalence, our goal is to show that for every assignment of x and y, the two sides will reduce to a pair of $\beta\eta$-equivalent terms.

3.1.1 Syntactic Equivalence

To make these ideas precise, we must properly define what is meant by term and equivalence of terms under $\approx$.

Let V be a set of variable symbols $(v_0, v_1, ...)$, then define the set $T_V^{\mathbb{N}_+}$ inductively,

$$T_V^{\mathbb{N}_+} ::= \bar{n} \mid v \mid \epsilon \mid t_1 \oplus t_2$$

where $n \in \mathbb{N}$, $v \in V$ and $t_1, t_2 \in T_V^{\mathbb{N}_+}$. Given this definition, we can interpret $T_V^{\mathbb{N}_+}$ as the set of (partially static) terms of the commutative monoid $\mathbb{N}_+ = (\mathbb{N}, +, 0)$, with variables in V. For example, the expression

$$(2 + x) + (y + 3) \tag{3.2}$$

corresponds to the term

$$(\bar{2} \oplus v_0) \oplus (v_1 \oplus \bar{3}) \tag{3.3}$$

Note that, when extracting the syntax of an expression, we must be careful to map any dynamic atomic sub-expressions to elements of V, and any static atomic sub-expressions to their inclusion in $T_V^{\mathbb{N}_+}$.

Given some context, $\Gamma : V \to \mathbb{N}$, assigning each variable symbol to an element of $\mathbb{N}$, we can define an interpretation function $[\![-]\!]_\Gamma^{\mathbb{N}_+} : T_V^{\mathbb{N}_+} \to \mathbb{N}$ recursively,

$$\begin{aligned}
[\![\bar{n}]\!]_\Gamma^{\mathbb{N}_+} &= n \\
[\![v]\!]_\Gamma^{\mathbb{N}_+} &= \Gamma(v) \\
[\![\epsilon]\!]_\Gamma^{\mathbb{N}_+} &= 0 \\
[\![t_1 \oplus t_2]\!]_\Gamma^{\mathbb{N}_+} &= [\![t_1]\!]_\Gamma^{\mathbb{N}_+} + [\![t_2]\!]_\Gamma^{\mathbb{N}_+}
\end{aligned}$$

As an example, let $\Gamma = [v_0 \mapsto x; v_1 \mapsto y]$. In this context, we can interpret the term

3.3 to recover 3.2,

$$\begin{aligned}
[\![(\bar{2} \oplus v_0) \oplus (v_1 \oplus \bar{3})]\!]_\Gamma^{\mathbb{N}_+} &= [\![\bar{2} \oplus v_0]\!]_\Gamma^{\mathbb{N}_+} + [\![v_1 \oplus \bar{3}]\!]_\Gamma^{\mathbb{N}_+} \\
&= ([\![\bar{2}]\!]_\Gamma^{\mathbb{N}_+} + [\![v_0]\!]_\Gamma^{\mathbb{N}_+}) + ([\![v_1]\!]_\Gamma^{\mathbb{N}_+} + [\![\bar{3}]\!]_\Gamma^{\mathbb{N}_+}) \\
&= (2 + \Gamma(v_0)) + (\Gamma(v_1) + 3) \\
&= (2 + x) + (y + 3)
\end{aligned}$$

With these definitions, we can now turn our attention to the equivalence $\approx$. In defining $\approx$, our goal is to relate precisely those terms that are provably equivalent up to associativity, commutativity, units and the evaluation of wholly static sub-terms. To this end, we may define $\approx$ inductively as follows,

$$\frac{}{t \approx t}\ (\textsc{Refl.}) \qquad \frac{t_1 \approx t_2}{t_2 \approx t_1}\ (\textsc{Sym.}) \qquad \frac{t_1 \approx t_2 \quad t_2 \approx t_3}{t_1 \approx t_3}\ (\textsc{Trans.})$$

$$\frac{t_1 \approx t_1' \quad t_2 \approx t_2'}{t_1 \oplus t_2 \approx t_1' \oplus t_2'}\ (\textsc{Cong.}) \qquad \frac{}{\bar{n}_1 \oplus \bar{n}_2 \approx \overline{n_1 + n_2}}\ (\oplus\text{-}\textsc{Eval.})$$

$$\frac{}{\epsilon \approx \bar{0}}\ (\epsilon\text{-}\textsc{Eval.}) \qquad \frac{}{\epsilon \oplus t \approx t}\ (\textsc{Unit}_l) \qquad \frac{}{t \oplus \epsilon \approx t}\ (\textsc{Unit}_r)$$

$$\frac{}{t_1 \oplus t_2 \approx t_2 \oplus t_1}\ (\textsc{Comm.}) \qquad \frac{}{(t_1 \oplus t_2) \oplus t_3 \approx t_1 \oplus (t_2 \oplus t_3)}\ (\textsc{Assoc.})$$

By including the rules Refl., Sym. and Trans., we ensure that this definition forms an equivalence over $T_V^{\mathbb{N}_+}$. The structural rules $\oplus$–Eval., ϵ–Eval. and Cong. allow us to compute within sub-terms, evaluating static sub-terms whenever possible. The remaining rules straight-forwardly codify the axioms of commutative monoids, introducing associativity, commutativity and ensuring ϵ is an identity for $\oplus$ (up to $\approx$).

Most importantly, using this definition of $\approx$, it can be shown that, given $t_1, t_2 \in T_V^{\mathbb{N}_+}$,

$$t_1 \approx t_2 \implies [\![t_1]\!]_\Gamma^{\mathbb{N}_+} = [\![t_2]\!]_\Gamma^{\mathbb{N}_+} \tag{3.4}$$

for any context Γ. That is, $[\![-]\!]_\Gamma^{\mathbb{N}_+}$ is a congruence over $\approx$.

The importance of this result becomes clear when considered in conjunction with the function $[-]_\approx : T_V^{\mathbb{N}_+} \to T_V^{\mathbb{N}_+}/\approx$, sending each $t \in T_V^{\mathbb{N}_+}$ to its equivalence class $[t]_\approx$. Given terms $t_1, t_2 \in T_V^{\mathbb{N}_+}$, for any context Γ,

$$\begin{aligned}
[t_1]_\approx = [t_2]_\approx &\iff t_1 \approx t_2 && \text{(by definition)} \\
&\implies [\![t_1]\!]_\Gamma^{\mathbb{N}_+} = [\![t_2]\!]_\Gamma^{\mathbb{N}_+} && \text{(using 3.4)}
\end{aligned}$$

In other words, we can reduce the problem of deriving algebraic identities to the problem of deriving identities between equivalence classes.

To give a concrete example, consider again the equation 3.1. The following diagram demonstrates the process of deriving this identity, making use of the results described above.

$$\begin{tikzcd}
{[(\bar{2} \oplus v_0) \oplus (v_1 \oplus \bar{3})]_\approx} \arrow[rr, equal] \arrow[dr, mapsto, "\in"] & & {[v_0 \oplus (v_1 \oplus \bar{5})]_\approx} \arrow[dl, mapsto, "\in"'] \\
(\bar{2} \oplus v_0) \oplus (v_1 \oplus \bar{3}) \arrow[u, mapsto, "{[-]_\approx}"] \arrow[d, mapsto, "{[\![-]\!]_\Gamma^{\mathbb{N}_+}}"] & (v_0 \oplus v_1) \oplus \bar{5} \arrow[d, mapsto, "{[\![-]\!]_\Gamma^{\mathbb{N}_+}}"'] & v_0 \oplus (v_1 \oplus \bar{5}) \arrow[u, mapsto, "{[-]_\approx}"'] \arrow[d, mapsto, "{[\![-]\!]_\Gamma^{\mathbb{N}_+}}"'] \\
(2+x)+(y+3) \arrow[r, equal] \arrow[u, dashed, bend left] & (x+y)+5 \arrow[r, equal] & x+(y+5) \arrow[u, dashed, bend right]
\end{tikzcd}$$

In particular, each side of the equation determines a piece of abstract syntax (illustrated with dashed arrows). As demonstrated, this syntax can be interpreted in the context $\Gamma = [v_0 \mapsto x; v_1 \mapsto y]$ to recover the original expressions. However, we can alternatively take each piece of syntax to its equivalence class under $\approx$. In this case, these equivalence classes are identical. From this fact and the inclusion $T_V^{\mathbb{N}_+}/\approx \subseteq T_V^{\mathbb{N}_+}$, we can *reify* a single piece of syntax, equivalent under $\approx$ to the two pieces of syntax we began with. It follows, using 3.4, that the interpretation of this single piece of syntax under Γ will be equal to both the left-hand side and right-hand side of the original equation. Thus, by transitivity, the original equation must hold.

3.1.2 Computing Equivalence Classes

Unfortunately, the problem of determining the equivalence class of an object under an arbitrary equivalence is undecidable. However, there are many specific equivalences which are decidable, with each equivalence class conveniently represented by a canonical representative. Doing so reduces the problem of deciding equality of equivalence classes to deciding equality of representatives, for which a naïve syntax-directed approach will suffice.

For the case of $T_V^{\mathbb{N}_+}$, notice that, as we are free to commute sub-terms, reassociate, eliminate instances of ϵ and reduce wholly static sub-terms, each term $t \in T_V^{\mathbb{N}_+}$ is equivalent to a term of the form

$$\left(\bigoplus_{v \in V} v^{k(v)} \right) \oplus \bar{n}$$

where

$$v^m = \underbrace{v \oplus \ldots \oplus v}_{m \text{ times}}$$

and $k : V \to \mathbb{N}$ sends each $v \in V$ to the number of occurrences of v in t. Notice that the left-hand side of this term is purely dynamic, and is determined entirely by k, while the right-hand is purely static, and is determined entirely by n.

This suggests that we have a correspondence between equivalence classes $[t]_{\approx}$ and pairs $(k, n) \in \mathbb{N}^V \times \mathbb{N}$. We can make this correspondence explicit by defining two maps:

$$\begin{aligned} \lfloor - \rfloor &: T_V^{\mathbb{N}_+} \to \mathbb{N}^V \times \mathbb{N} \ (\mathit{reflect}) \\ \lceil - \rceil &: \mathbb{N}^V \times \mathbb{N} \to T_V^{\mathbb{N}_+} \ (\mathit{reify}) \end{aligned}$$

such that, for all $t_1, t_2 \in T_V^{\mathbb{N}_+}$ and $p_1, p_2 \in \mathbb{N}^V \times \mathbb{N}$,

$$t_1 \approx t_2 \iff \lfloor t_1 \rfloor = \lfloor t_2 \rfloor \tag{3.5}$$

and

$$\lceil p_1 \rceil = \lceil p_2 \rceil \iff p_1 = p_2$$

It is not difficult to see that, to this end, we can define $\lfloor - \rfloor$ as

$$\begin{aligned} \lfloor \bar{n} \rfloor &= ([\_ \mapsto 0], n) \\ \lfloor v \rfloor &= ([v \mapsto 1; \_ \mapsto 0], 0) \\ \lfloor \epsilon \rfloor &= ([\_ \mapsto 0], 0) \\ \lfloor t_1 \oplus t_2 \rfloor &= \lfloor t_1 \rfloor \boxplus \lfloor t_2 \rfloor \end{aligned}$$

where

$$(k_1, n_1) \boxplus (k_2, n_2) = (\lambda x.y.k_1(x) + k_2(y), n_1 + n_2)$$

and $\lceil - \rceil$ as

$$\lceil (k, n) \rceil = \left(\bigoplus_{v \in V} v^{k(v)} \right) \oplus \bar{n}$$

With these maps, it finally becomes possible to derive a proof of 3.1 mechanically. In particular, reflecting the syntax of each side of the equation, we find that

$$\begin{aligned}
\left\lfloor(\bar{2} \oplus v_0) \oplus (v_1 \oplus \bar{3})\right\rfloor &= \left\lfloor \bar{2} \oplus v_0 \right\rfloor \boxplus \left\lfloor v_1 \oplus \bar{3} \right\rfloor \\
&= (\left\lfloor \bar{2} \right\rfloor \boxplus \lfloor v_0 \rfloor) \boxplus (\lfloor v_1 \rfloor \boxplus \left\lfloor \bar{3} \right\rfloor) \\
&= (([v_0 \mapsto 0; v_1 \mapsto 0], 2) \boxplus ([v_0 \mapsto 1; v_1 \mapsto 0], 0)) \\
&\quad \boxplus (([v_0 \mapsto 0; v_1 \mapsto 1], 0) \boxplus ([v_0 \mapsto 0; v_1 \mapsto 0], 3)) \\
&= ([v_0 \mapsto 1; v_1 \mapsto 0], 2) \boxplus ([v_0 \mapsto 0; v_1 \mapsto 1], 3) \\
&= ([v_0 \mapsto 1; v_1 \mapsto 1], 5)
\end{aligned}$$

and

$$\begin{aligned}
\left\lfloor v_0 \oplus (v_1 \oplus \bar{5})\right\rfloor &= \lfloor v_0 \rfloor \boxplus \left\lfloor v_1 \oplus \bar{5} \right\rfloor \\
&= \lfloor v_0 \rfloor \boxplus (\lfloor v_1 \rfloor \boxplus \left\lfloor \bar{5} \right\rfloor) \\
&= ([v_0 \mapsto 1; v_1 \mapsto 0], 0) \\
&\quad \boxplus (([v_0 \mapsto 0; v_1 \mapsto 1], 0) \boxplus ([v_0 \mapsto 0; v_1 \mapsto 0], 5)) \\
&= ([v_0 \mapsto 1; v_1 \mapsto 0], 0) \boxplus ([v_0 \mapsto 0; v_1 \mapsto 1], 5) \\
&= ([v_0 \mapsto 1; v_1 \mapsto 1], 5)
\end{aligned}$$

Thus, by 3.5 and 3.4, the original equation must hold. We can also reify this canonical representation to obtain a *normal form*,

$$\lceil ([v_0 \mapsto 1; v_1 \mapsto 1], 5) \rceil = (v_0 \oplus v_1) + 5$$

It should be noted, at this point, that none of the techniques described so far in this chapter are unique to $\mathbb{N}_+$, or even commutative monoids in general. While the details of the constructions differ, this precise approach to deriving equivalences is implemented for a number of algebraic structures in Agda's standard library. These structures include rings, monoids and commutative monoids. For each, a set of terms is defined alongside a notion of syntactic equivalence. A method for computing representatives for equivalence classes is then stated and verified, ultimately yielding a proof synthesis solution similar to that described here.

However, the goal of this project is to develop a general framework, within which algebraic solvers for arbitrary structures can be implemented and verified. It is therefore essential identify the shared properties of these constructions, with the aim of describing a process by which their specifications can be derived automatically, given some algebraic theory.

3.1.3 A Universal Property

The objects described so far in this chapter fulfill deep roles in the category **CMon**, of commutative monoids and monoid homomorphisms. The analogues of these structures in other categories of algebras satisfy the same key universal property. Critically, it is precisely this universal property that enables the proof synthesis approach described in this chapter. Thus, the goal of this section is to explain how we arrive at this universal property, and what this means in terms of the specification of algebraic solvers.

First, observe that $T_V^{\mathbb{N}_+}$ forms a commutative monoid under $\oplus$, working up to $\approx$. By quotienting appropriately, we therefore obtain a commutative monoid $(T_V^{\mathbb{N}_+}/\approx, \oplus, \epsilon)$. From this, it is trivial to show that, for any context Γ, $[\![-]\!]_\Gamma^{\mathbb{N}_+}$ lifts to a homomorphism, $[\![-]\!]_\Gamma^{\mathbb{N}_+} : (T_V^{\mathbb{N}_+}/\approx, \oplus, \epsilon) \to \mathbb{N}_+$.

This idea generalises: given an arbitrary commutative monoid $X = (X, \otimes, u)$, a homomorphism $f : \mathbb{N}_+ \to X$ and a context $\Gamma : V \to X$, we obtain a homomorphism $[\![-]\!]_{\Gamma,f}^X : (T_V^{\mathbb{N}_+}/\approx, \oplus, \epsilon) \to X$ given by,

$$\begin{aligned}
[\![\bar{n}]\!]_{\Gamma,f}^X &= f(n) \\
[\![v]\!]_{\Gamma,f}^X &= \Gamma(v) \\
[\![\epsilon]\!]_{\Gamma,f}^X &= u \\
[\![t_1 \oplus t_2]\!]_{\Gamma,f}^X &= [\![t_1]\!]_{\Gamma,f}^X \otimes [\![t_2]\!]_{\Gamma,f}^X
\end{aligned}$$

In fact, it is not difficult to show that for a given Γ and f, this is the *unique* such homomorphism.

Let $T_V \subseteq T_V^{\mathbb{N}_+}$ be the set of purely dynamic terms over V. Note that $(T_V/\approx, \oplus, \epsilon)$ too forms a commutative monoid. However, this monoid has the special property that, for any commutative monoid $X = (X, \otimes, u)$, each homomorphism $(T_V/\approx, \oplus, \epsilon) \to X$ is uniquely determined by a context $\Gamma : V \to X$. Given this property, we say that $(T_V/\approx, \oplus, \epsilon)$ is the *free* commutative monoid generated by V.

Thus, given a context $\Gamma : V \to X$ and a homomorphism $f : \mathbb{N}_+ \to X$, the uniqueness of $[\![-]\!]_{\Gamma,f}^X$ can be illustrated diagrammatically as follows,

$$\begin{tikzcd}
& X & \\
\mathbb{N}_+ \arrow[ur, "f"] \arrow[r, hook, "\bar{(-)}"'] & (T_V^{\mathbb{N}_+}/\approx, \oplus, \epsilon) \arrow[u, dashed, "\exists! [\![-]\!]_{\Gamma,f}^X"'] & (T_V/\approx, \oplus, \epsilon) \arrow[l, hook', "\subseteq"] \arrow[ul, "{[\![-]\!]_\Gamma^X}"']
\end{tikzcd}$$

The reader may recognise this as the familiar coproduct in the category **CMon**. It follows that $(T_V^{\mathbb{N}_+}/\approx, \oplus, \epsilon)$ is simply the binary coproduct of $\mathbb{N}_+$ with the free commutative monoid over V.

Turning our attention to $\mathbb{N}^V \times \mathbb{N}$, we again discover a commutative monoid: $(\mathbb{N}^V \times \mathbb{N}, \boxplus, \lfloor \epsilon \rfloor)$. Moreover, as $\lfloor - \rfloor$ and $\lceil - \rceil$ trivially lift to homomorphisms, we have an isomorphism,

$$\begin{tikzcd}
(T_V^{\mathbb{N}_+}/\approx, \oplus, \epsilon) \arrow[r, shift left, "\lfloor - \rfloor"] & (\mathbb{N}^V \times \mathbb{N}, \boxplus, \lfloor \epsilon \rfloor) \arrow[l, shift left, "\lceil - \rceil"]
\end{tikzcd}$$

It is a standard result of category theory that objects isomorphic to a (co)-limit object enjoy the same universal properties. In this case, this means that the algebra $(\mathbb{N}^V \times \mathbb{N}, \boxplus, \lfloor \epsilon \rfloor)$ also forms a binary coproduct of $\mathbb{N}_+$ and $(T_V/\approx, \oplus, \epsilon)$ – or, diagrammatically,

$$\begin{tikzcd}
 & X & \\
\mathbb{N}_+ \arrow[ur, "f"] \arrow[r, "{([\_ \mapsto 0], -)}"'] & (\mathbb{N}^V \times \mathbb{N}, \boxplus, \lfloor \epsilon \rfloor) \arrow[u, dashed, "\exists ! h"'] & (T_V/\approx, \oplus, \epsilon) \arrow[ul, "{[\![-]\!]_\Gamma^X}"'] \arrow[l, "\lfloor - \rfloor"]
\end{tikzcd}$$

However, we can work backwards: any two (co)-limits for the same diagram are provably isomorphic. Thus, supposing we have an arbitrary coproduct of $\mathbb{N}_+$ with the free commutative monoid over V, we immediately obtain an isomorphism with $(T_V^{\mathbb{N}_+}/\approx, \oplus, \epsilon)$. Moreover, as this result holds constructively, we can actually compute this isomorphism. Thus, given such a coproduct, we can mechanically derive its corresponding reflection and reification maps and proofs that they satisfy all of the properties needed for proof synthesis.

Again, none of the observations made here are specific to $\mathbb{N}_+$ or **CMon**. Informally, if we have some general equational theory Θ such that its category of models $\mathbf{Alg}_\Theta$ supports the construction of free models, so long as we can construct coproducts with these free models, proof synthesis becomes possible. Hence, somewhat imprecisely, constructing an algebraic solver for Θ reduces to providing a method for constructing a coproduct of an arbitrary model $A \in \mathbf{Alg}_\Theta$ with the free model generated by some base object V. We call this coproduct $A[V]$, read *'the free extension of A by V'*. The remainder of this chapter is dedicated to making this idea precise, for a broad class of equational systems.

3.2 Universal Algebra & Equational Systems

Informally, universal (sometimes 'abstract') algebra is the the study of algebraic structures in an abstract setting. That is, universal algebra gives us a language with which to reason about algebraic structures (e.g., groups, rings and fields) without consideration for the particular properties of any one such structure. General results in universal algebra therefore apply to *all* such structures. Abstract algebras are sometimes characterised as 'sets with operations'; while appealing, this view is too limited since free extensions are applicable to a much broader class of equational systems introduced by Fiore & Hur[8].

3.2.1 Algebraic Signatures

Fundamental to universal algebra is the notion of an *algebraic signature*. Just as programmers think of programs as having types, algebraists see algebras as having signatures. Extending this analogy, just as a type describes the structure of its elements, an algebraic signature describes the structure of members of a class of algebras.

Under a set-theoretic worldview, a (mono-sorted) algebraic signature $\Sigma = (\mathcal{O}, [-])$ consists of a set $\mathcal{O}$, the set of operator symbols, and a function $[-] : \mathcal{O} \to \mathbb{N}$, assigning each operator symbol its arity. Given an algebraic signature Σ, a Σ-*algebra* $X = (X, [\![-]\!]_X)$ then consists of a set X, the *carrier set*, and an $\mathcal{O}$-indexed family of functions $[\![-]\!]_X$ sending each $f \in \mathcal{O}$ to an $[f]$-ary function $[\![f]\!]_X : X^{[f]} \to X$, the *interpretation of* f.

It is well known that this is equivalent to providing a single function

$$[\![-]\!]_X : \coprod_{f \in \mathcal{O}} X^{[f]} \to X$$

That is, a function from a polynomial in X into X, where the nature of the polynomial is specified entirely by the signature. In fact, given any algebraic signature Σ, there is a polynomial endofunctor $\bar{\Sigma} : \mathbf{Set} \to \mathbf{Set}$ given by

$$\bar{\Sigma}(S) = \coprod_{f \in \mathcal{O}} S^{[f]}$$

Making this observation, defining a Σ-algebra reduces to identifying a carrier set X and constructing a function $[\![-]\!]_X : \bar{\Sigma}X \to X$.

The critical observation that Fiore & Hur make is that this notion of algebraic signature is neither limited to the category of sets, nor to polynomial endofunctors.

Thus, in a general, categorical setting, the notion of an algebraic signature over a base category reduces simply to an endofunctor on that category.

Definition 1. A *functorical signature* on a category $\mathcal{C}$ is an endofunctor $\Sigma : \mathcal{C} \to \mathcal{C}$.

Definition 2. A Σ-*algebra* for a functorial signature Σ is a pair (X, s) of an object $X \in \mathcal{C}$ and a map $s : \Sigma X \to X \in \mathcal{C}$.

3.2.2 The Category of Σ-Algebras

Familiar from mathematics are the notions of homomorphisms of monoids, groups and rings. From a set-theoretic perspective, these are functions between carrier sets that, in some technical sense, preserve the underlying algebraic structure. For a concrete example, consider the monoids $A = (A, \otimes_A, e_A)$ and $B = (B, \otimes_B, e_B)$. Taking some function $h : A \to B \in \mathbf{Set}$, we say that h is a *homomorphism (of monoids)* so long as

$$h(e_A) = e_B$$

and for all $x, y \in A$,

$$h(x \otimes_A y) = h(x) \otimes_B h(y)$$

This definition generalises neatly to our set-theoretic notion of an algebraic signature. In particular, given an algebraic signature $\Sigma = (\mathcal{O}, [-])$, Σ-algebras $X = (X, [\![-]\!]_X)$ & $Y = (Y, [\![-]\!]_Y)$, and a function $h : X \to Y \in \mathbf{Set}$, we say that h is a *homomorphism (of* Σ*-algebras)* if, for all $f \in \mathcal{O}$ and $x_i \in X$,

$$h([\![f]\!]_X(x_1, ..., x_{[f]})) = [\![f]\!]_Y(h(x_1), ..., h(x_{[f]}))$$

Notice that this is equivalent to asserting that

$$h \circ [\![-]\!]_X = [\![-]\!]_Y \circ \bar{\Sigma} h$$

Lifting this condition to functorial signatures gives rise to the following categorical notion of a homomorphism of algebras.

Definition 3. A *homomorphism (of* Σ*-algebras)* $(X, s) \to (Y, t)$ is a map $h : X \to Y$ with the property that the following commutes as a diagram in $\mathcal{C}$.

$$\begin{tikzcd}
\Sigma X \arrow[r, "s"] \arrow[d, "\Sigma h"'] & X \arrow[d, "h"] \\
\Sigma Y \arrow[r, "t"'] & Y
\end{tikzcd}$$

It is quick to verify that this definition of homomorphism inherits identities and composition from $\mathcal{C}$. Thus, we obtain a category $\mathbf{Alg}_\Sigma$ of Σ-algebras and their homomorphisms. Observe that we always have a functor $U_\Sigma : \mathbf{Alg}_\Sigma \to \mathcal{C}$ (the *forgetful functor*) mapping each Σ-algebra (X, s) to its carrier object X.

3.2.3 Equational Systems

When defining an equational theory Θ, such as that of groups, we do so by providing a signature Σ alongside a set of (first-order) equations, which any model of this theory is expected to satisfy. Such equations are presented as pairs of terms over Σ, containing variables drawn from a set V (i.e., elements of the set $T_\Sigma(V)$). A model of Θ is then a Σ-algebra X for which any evaluation of such a pair yields an identity. To reflect this categorically, Fiore & Hur develop a general, categorical notion of term, from which we can derive notions of equation and satisfaction.

In particular, Fiore & Hur observe that, in a set-theoretic setting, given some Σ-algebra $(X, [\![-]\!]_X)$, any term $t \in T_\Sigma(V)$ determines a map $[\![t]\!]_X : X^V \to X$, sending each context γ to the interpretation of t under γ in X. However, as $\Gamma_V = (-)^V : \mathbf{Set} \to \mathbf{Set}$ is itself a polynomial endofunctor, we have determined a new algebraic signature Γ_V, and a Γ_V-algebra $(X, [\![t]\!]_X)$.

This gives us an interesting perspective on the nature of contexts. Specifically, given $(X, [\![-]\!]_X)$, $\Gamma_V X$ determines the set of all X-valued contexts over V. Furthermore, given a homomorphism $h : (X, [\![-]\!]_X) \to (Y, [\![-]\!]_Y)$, we interpret $\Gamma_V h$ as the function sending each context $\gamma \in \Gamma_V X$ to the context $h \circ \gamma \in \Gamma_V Y$. From this, it should be clear that the following will commute,

$$\begin{tikzcd}
\Gamma_V X \arrow[r, "{[\![t]\!]_X}"] \arrow[d, "\Gamma_V h"'] & X \arrow[d, "h"] \\
\Gamma_V Y \arrow[r, "{[\![t]\!]_Y}"'] & Y
\end{tikzcd}$$

Or, equivalently, any homomorphism of Σ-algebras is also a homomorphism of Γ_V-

algebras. Thus, any term $t \in T_\Sigma(V)$ determines a functor $\bar{t} : \mathbf{Alg}_\Sigma \to \mathbf{Alg}_{\Gamma_V}$, sending each $(X, [\![-]\!]_X)$ to $(X, [\![t]\!]_X)$ while preserving carrier objects and homomorphisms. Working backwards from here, Fiore & Hur arrive at the following categorical notions of term and equational system.

Definition 4. A *functorial term* $\mathcal{C} : \Sigma \rhd \Gamma \vdash T$ consists of an endofunctor Γ on $\mathcal{C}$ (the *functorial context*) and a functor $T : \mathbf{Alg}_\Sigma \to \mathbf{Alg}_\Gamma$ such that $U_\Gamma \circ T = U_\Sigma$ (i.e., T preserves carrier objects and homomorphisms).

Definition 5. An *equational system* $\Theta = (\mathcal{C} : \Sigma \rhd \Gamma \vdash L = R)$ is a parallel pair of functorial terms $L, R : \mathbf{Alg}_\Sigma \to \mathbf{Alg}_\Gamma$ over a base category $\mathcal{C}$.

Note, while it appears that this definition only captures equational systems with a single equation, Fiore & Hur demonstrate that any system of equations can be captured as a single pair of functorial terms (so long as $\mathcal{C}$ supports indexed coproducts).

Definition 6. A *model* of an equational system $\Theta = (\mathcal{C} : \Sigma \rhd \Gamma \vdash L = R)$ is any Σ-algebra (X, s) such that $L(X, s) = R(X, s)$.

Taking the equaliser of L and R, we obtain the category of Θ-models $\mathbf{Alg}_\Theta$,

$$\begin{tikzcd}
\mathbf{Alg}_\Theta \arrow[r, hook, "j_\Theta"] & \mathbf{Alg}_\Sigma \arrow[r, shift left, "L"] \arrow[r, shift right, "R"'] & \mathbf{Alg}_\Gamma \\
\mathcal{D} \arrow[u, dashed, "\exists! \bar{F}"] \arrow[ur, "F"'] & &
\end{tikzcd}$$

Critically, $\mathbf{Alg}_\Theta$ is a sub-category of $\mathbf{Alg}_\Sigma$, meaning that we can treat homomorphisms of algebras and models uniformly. Moreover, $\mathbf{Alg}_\Theta$ inherits a forgetful functor $U_\Theta = U_\Sigma \circ j_\Theta : \mathbf{Alg}_\Theta \to \mathcal{C}$, obtained by precomposing U_Σ onto the embedding j_Θ.

3.2.4 Free Algebras

Another key idea from universal algebra is the notion of a *free algebra*. From a set-theoretic point of view, the free Σ-algebra over a set of generators V is the set of terms $T_\Sigma(V)$, equipped with the algebraic structure it inherits from its inductive construction. However, the key property that free algebras enjoy is that each map from their set of generators to the carrier of another algebra induces a unique homomorphism. Hence, the following categorical generalisation.

Definition 7. Given an object $X \in \mathcal{C}$, a *free Σ-algebra over X* is any algebra $X' \in \mathbf{Alg}_\Sigma$ equipped with a map $\eta : X \to U_\Sigma X'$, such that, for all $Y \in \mathbf{Alg}_\Sigma$ and $f : X \to U_\Sigma Y \in \mathcal{C}$, there exists a unique homomorphism $\bar{f} : X' \to Y$ with $U_\Sigma \bar{f} \circ \eta = f$.

The following diagram makes this explicit,

$$\begin{tikzcd}
X' \arrow[d, dashed, "\exists! \bar{f}"'] & U_\Sigma X' \arrow[d, "U_\Sigma \bar{f}"'] & X \arrow[l, "\eta"'] \arrow[dl, "f"] \\
Y & U_\Sigma Y &
\end{tikzcd}$$

It is not difficult to see that by naturalising this picture in X, renaming X' to $F_\Sigma X$, we obtain the characterisation of a left adjoint to U_Σ.

$$\begin{tikzcd}
F_\Sigma X \arrow[d, dashed, "\exists! \bar{f}"'] & U_\Sigma(F_\Sigma X) \arrow[d, "U_\Sigma \bar{f}"'] & X \arrow[l, "\eta_X"'] \arrow[dl, "f"] \\
Y & U_\Sigma Y &
\end{tikzcd}$$

Thus, a free Σ-algebra over X, if one exists, is unique up to isomorphism and $\mathbf{Alg}_\Sigma$ has all free Σ-algebras precisely when U_Σ has a left adjoint F_Σ. It is worth noting that this adjunction generalises the the familiar free-forgetful adjunctions enjoyed by algebraic structures in **Set**.

3.2.5 Free Models

From this notion of a free algebra we can further derive the notion of a *free model*. In set-theoretic universal algebra, free models of a theory can be obtained by taking the quotient of a free algebra by the equivalence on terms induced by the theory's equational logic. However, in a categorical setting, the realisation that the essential properties of free algebras follow from the universal property they satisfy leads to a direct definition of free models by way of extending the adjunction $F_\Sigma \dashv U_\Sigma$ to U_Θ

as pictured,

$$\begin{tikzcd}
\mathbf{Alg}_\Theta \arrow[r, hook, shift right, "j_\Theta"'] \arrow[r, phantom, "\perp"] \arrow[rr, bend right=40, "U_\Theta = U_\Sigma \circ j_\Theta"'] & \mathbf{Alg}_\Sigma \arrow[l, dashed, shift right, "r_\Theta"'] \arrow[r, shift right, "U_\Sigma"'] \arrow[r, phantom, "\perp"] & \mathcal{C} \arrow[l, shift right, "F_\Sigma"'] \arrow[ll, bend right=40, "F_\Theta = r_\Theta \circ F_\Sigma"']
\end{tikzcd}$$

It is a standard result in category theory that the left-adjoint to an inclusion is a quotient map, thus preserving our intuition from equational theories defined over **Set**.

3.3 Free Extensions

Using the definitions given in §3.2, we can now give a precise definition of free extensions for an equational system.

Definition 8. In an equational system $\Theta = (\mathcal{C} : \Sigma \rhd \Gamma \vdash L = R)$, given $V \in \mathcal{C}$, the *free extension* of $X \in \mathbf{Alg}_\Theta$ by V, if it exists, is the coproduct $X[V] = X + F_\Theta V$, where $F_\Theta \dashv U_\Theta$.

From this definition, it follows that whenever $\mathbf{Alg}_\Theta$ has binary coproducts, if U_Θ has a left adjoint, $(-)[=] : \mathbf{Alg}_\Theta \times \mathcal{C} \to \mathbf{Alg}_\Theta$ lifts to a bifunctor. In such situations, we say that '$\mathbf{Alg}_\Theta$ *has all free extensions*'. It is also worth noting that, when authoring a solver for Θ, it is this bifunctor that we will need to provide.

3.3.1 Normalisation by Evaluation

At this point, it is reasonable to question the motivation for defining free extensions in this abstract setting. However, doing so not only introduces free extensions to the world beyond simple first-order equational theories, but further suggests deep connections between free extensions and more studied techniques, such as *normalisation by evaluation* (NbE).

The applications of NbE are far-reaching, with notable examples including deciding the word problem for simply-typed λ-calculus with coproducts[2], and computing isomorphisms of recursive polynomial types[7]. Broadly speaking, NbE is a family of techniques for discovering normal forms in various λ-calculi. Rather than rewriting, under NbE, terms are reflected into a non-standard denotational semantics in order to determine normal forms. Once found, normal forms are then reified, recovering equivalent canonical terms.

From this informal description, a connection between NbE techniques and free extensions is clear: both enable normalisation in a family of equational systems by reflecting syntax into a non-standard denotational semantics and then reifying provably equivalent syntax. Despite this, the connection between NbE and free extensions is yet to be seriously acknowledged, as free extensions have, thus far, only been applied to first-order equational theories which cannot encode the variable binding constructs found in λ-calculi.

However, the class of equational systems discussed in this chapter *is* broad enough to capture a variety of λ-calculi. In particular, these definitions cover the second-order equational logics introduced by Fiore & Hur in [9], and further developed by Fiore & Mahmoud in [10]. Unlike first-order theories, second-order theories can support variable-binding constructs, and can therefore be used to present theories such as the simply-typed λ-calculus – or, equivalently, the equational theory of cartesian-closed categories.

As such, this work conjectures that existing NbE techniques can be reinterpreted as free extensions in the category of cartesian-closed categories (or similar). However, constructing specific instances of this correspondence, and investigating the extent to which free extensions generalise NbE techniques is left as future work.

3.4 Summary

To summarise, this chapter demonstrated that a frequently exploited approach to deriving algebraic identities led naturally to a universal property in the category of models of an equational theory. Specifically, we saw that algebras of normal forms are, in general, coproducts with free models (free extensions).

Building on this, the latter half of this chapter described a broad class of equational systems, introduced by Fiore & Hur, to which the notion of free extensions generalises. In doing so, this chapter developed a strong theoretical basis, serving as the foundation for the formalisation described in the next chapter. Moreover, taking free extensions into this abstract setting pointed to a deep connection between normalisation-by-evaluation techniques and free extensions for second-order equational theories. As such, it is important to recognise that while the coming chapter makes a number of simplifying assumptions as to the family of equational theories it considers, this is not a consequence of a theoretical limitation of the techniques described.

4 — Formalisation

This chapter describes a formalisation of free extensions in the Agda proof assistant. While the definition of free extensions given in the previous chapter covers a variety of exotic equational systems, this formalisation is limited to finitary, mono-sorted algebraic structures (i.e., structures built from a collection of untyped operators of finite arity). However, as illustrated by the previous chapter, this restriction is not a consequence of a theoretical limitation of free extensions, but was adopted from the outset to manage this project's complexity, with generalisations left as future work.

Overview §4.1 states and justifies the key design decisions made when beginning the development of the formalisation described in this chapter. These decisions ultimately led to the need for (and implementation of) a lightweight formalisation of universal algebra, tailored to suit this project. §4.2 presents the algebraic component of this formalisation, while §4.3 discusses its equational aspects. Building on this foundation, §4.4 describes the formal specification of free extensions, stating the key results enabling proof synthesis. Finally, §4.5 concludes with a brief summary.

4.1 Design Decisions

This formalisation adopts Agda's '`--without-K --safe --exact-split`' configuration. This is a strict configuration which enforces strong guarantees. The first flag, '`--without-K`', disables the type-checking rules that would make the axiom K typeable, thus guaranteeing intensionality. The second flag '`--safe`', disables a host of features which can introduce inconsistencies to Agda's logic, such as postulates. The third flag, '`--exact-split`', ensures that all pattern matching holds definitionally. As the formalisation described here serves as the foundation for a broadly applicable tactic, it is required to be compatible with a range of Agda configurations. However, Hu & Carette observe in [13] that '`--without-K --safe`' is a widely compatible configuration, as it retains compatibility with both the '`--with-K`' (propositionally extensional) and '--cubical' (CuTT) Agda variants.

When beginning this work, I investigated existing formalisations of universal algebra with the hope of reusing them: in particular, DeMeo's UALib[5] and Gunther *et al.*'s formalisation[11]. However, DeMeo's reliance on a large, unstable Agda library (namely, Escardó's TypeTopology[6]) made it unsuitable from a compatibility perspective. Conversely, while Gunther *et al.*'s formalisation depends exclusively on Agda's standard library, it goes to great lengths to capture multi-sorted algebras and conditional-equational theories in full generality, making it unnecessarily difficult to work with when specialising to mono-sorted equational theories. Thus, I settled on building a ground-up formalisation of the fragment of universal algebra required to specify finitary, mono-sorted free extensions, relying only on Agda's standard library.

To support the *specification* of free extensions (i.e., coproducts with free models), an algebra formalisation must provide a construction of the free model of an arbitrary equational theory. However, as discussed in §3.2.4, this requires us to provide a left adjoint to the forgetful functor. While the category of sets admits this construction via the quotient of a free algebra by a theory's equational logic, the category of MLTT types does not admit arbitrary quotients, making a direct construction impossible. Thus, rather than defining algebraic structures over MLTT types directly, I define all structures over setoids (i.e., sets equipped with an equivalence relation – '*book equality*'[3]), as is done in Agda's standard library; thus, allowing quotients to be emulated with a change of equivalence.

The final design decision relates to the hierarchy of universes described in §2.1.5. In particular, many interesting algebraic structures inhabit universes other than $\mathcal{U}_0$. For example, the cartesian product of types, $A \times B$, in any universe $\mathcal{U}_i$ forms a commutative semigroup up to equivalence of types, e.g., $A \times B \simeq B \times A$. Synthesising proofs of these type equivalences is a compelling application of the proof synthesis technique described by chapter 3. Thus, to avoid repeatedly formalising algebraic structures in higher and higher universes, I elected to make definitions my *universe polymorphic*. Agda supports universe polymorphism by allowing quantification over *universe levels* (the ordinal indices of universes), but places universe-polymorphic types in transfinite universes which cannot be quantified over, to avoid the paradoxes touched on in §2.1.5.

4.2 Universal Algebra

This section describes my formalisation of universal algebra. The majority of the core definitions are mono-sorted specialisations of those given by Gunther *et al.*, but there are key differences in the presentation of free algebras. These definitions are mathematically less elegant than Gunther *et al.*'s, but greatly simplify proofs involving free constructions and are therefore a better fit for this use case.

4.2.1 Signatures

Under the assumption that operators are finitary and mono-sorted, an algebraic signature is characterised by a set of operator symbols, indexed by their respective arities,

```
record Signature : Set₁ where
  field
    ops : ℕ → Set
```

where Set_i denotes the universe $\mathcal{U}_i$ (with Set shorthand for Set_0). This characterisation of algebraic signatures is a mono-sorted specialisation of that used by Gunther *et al.* and allows for convenient pattern-matching on the operators of a signature.

As a concrete example, the signature of a magma (i.e., a single binary operator) is encoded as follows,

```
data MagmaOp : ℕ → Set where
  • : MagmaOp 2

Σ-magma : Signature
Σ-magma = record { ops = MagmaOp }
```

4.2.2 Interpretations

Given an algebraic signature Σ, an interpretation of Σ in the set A is a function specifying the action of each of Σ's operators on a vector whose length is the operator's arity,

```
Interpretation : Set a
Interpretation = ∀ {arity} → (f : ops Σ arity) → Vec A arity → A
```

Here, *a* is a *level parameter*. Level parameters allow for quantification over the hierarchy of universes, thus enabling the universe polymorphism discussed in §4.1.

An interpretation is a *congruence* over an equivalence so long as, given pointwise-equivalent inputs, it yields equivalent outputs,

```
Congruence : Interpretation → Set (a ⊔ ℓ)
Congruence ⟦_⟧ = ∀ {arity}
                   → (f : ops Σ arity)
                   → ∀ {xs ys} → Pointwise _≈_ xs ys → ⟦ f ⟧ xs ≈ ⟦ f ⟧ ys
```

4.2.3 Algebras

A setoid forms an algebra whenever we can equip it with an interpretation that forms a congruence over this setoid's book equality,

```
record IsAlgebra (S : Setoid a ℓ) : Set (a ⊔ ℓ) where
  field
    ⟦_⟧ : Interpretation S
    ⟦⟧-cong : Congruence S ⟦_⟧
```

The following definition is used to conveniently bundle a setoid with a proof that it forms an algebra,

```
record Algebra : Set (suc a ⊔ suc ℓ) where
  constructor algebra
  field
    ||_||/≈ : Setoid a ℓ
    ||_||/≈-isAlgebra : IsAlgebra ||_||/≈
```

When working with setoids, definitions must be polymorphic in both the universe of carrier sets and the universe in which book equalities are defined, hence the need for two level parameters a and ℓ.

4.2.4 Homomorphisms

A function between carrier sets of algebras is said to be *homomorphic* whenever it commutes with interpretation up to book equality in the algebra over its codomain,

```
Homomorphic : (S : Algebra {a} {ℓ₁}) (T : Algebra {b} {ℓ₂})
                    → (|| S || → || T ||) → Set (a ⊔ ℓ₂)
Homomorphic S T h = ∀ {arity} → (f : ops Σ arity)
                          → (xs : Vec || S || arity)
                          → T ⟦ f ⟧ (map h xs) =[ T ] h (S ⟦ f ⟧ xs)
```

A homomorphism of algebras is therefore any setoid morphism (i.e., congruence – written _↝_) between carriers that is also homomorphic,

```
record _⇝_ : Set (a ⊔ b ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    |_|⃗ : || A ||/≈ ↝ || B ||/≈
    |_|-hom : Homomorphic A B (Morphism.|_| |_|⃗)
```

Identities and composition are defined in the obvious way, with associativity and unit laws proven pointwise up to book equality in the codomain of composites. Similarly, equivalences of algebras are defined as pairs of mutually inverse homomorphisms, up to pointwise book equality with the identity homomorphism.

4.2.5 Free Algebras

The set of Σ-terms over a set A is the closure of A under well-formed applications of Σ's operators,

```
data Term (A : Set a) : Set a where
  atom : A → Term A
  term : ∀ {arity} → (f : ops Σ arity) → Vec (Term A) arity → Term A
```

Given an equivalence over A, we also obtain an equivalence of terms, given pointwise,

```
data _~_ : Term A → Term A → Set (a ⊔ ℓ) where
  atom : ∀ {x y} → x ≈ y → atom x ~ atom y
  term : ∀ {arity xs ys} {f : ops Σ arity}
        → Pointwise _~_ xs ys
        → term f xs ~ term f ys
```

Thus, given a setoid with carrier A, the Herbrand universe generated by this setoid becomes,

```
Herbrand : Setoid _ _
Herbrand = record { Carrier = Term A
                  ; _≈_     = _~_
                  ; isEquivalence = ~-isEquivalence
                  }
```

Moreover, given a setoid S, it is trivial to prove that Herbrand S forms a Σ-algebra, thus yielding a functor Free sending a setoid to its corresponding free algebra.

However, we are generally interested in algebras of terms containing some number of free variables. As all operators are finitary, we only ever have use for finite sets of variables. Hence, we can assume without loss of generality that variables are finite numerals up to some maximum n. Thus, the type BT of *possibly-static* elements of a set A becomes,

```
data BT (A : Set a) (n : ℕ) : Set a where
  sta : A → BT A n
  dyn : Fin n → BT A n
```

where sta marks static elements (i.e., elements of A), and dyn marks dynamic elements (i.e., variables). Given _≃_, an equivalence on A, we also obtain an equivalence over elements of BT A n,

```
data _≍_ : BT A n → BT A n → Set (a ⊔ ℓ) where
  sta : ∀ {x y} → x ≈ y → sta x ≍ sta y
  dyn : ∀ {x y} → x ≡ y → dyn x ≍ dyn y
```

This determines the setoid of atomic terms over a setoid S,

```
Atoms : Setoid a (a ⊔ ℓ)
Atoms = record { Carrier = BT (Setoid.Carrier S) n
               ; _≈_       = _≍_
               ; isEquivalence = ≍-isEquivalence
               }
```

As such, algebras of terms containing free variables become free algebras over a setoid of atoms.

As shorthand, F n denotes the finitely generated algebra with n generators. To allow for definitions made for arbitrary algebras of atoms to be reused for finitely generated algebras, I define F as follows,

```
F : ℕ → Algebra
F = Free ∘ Atoms (PE.setoid ⊥)
```

where PE.setoid sends a set to the setoid it forms under _≡_, and ⊥ denotes the empty type.

Given the understanding of variables described above, an environment (context) over an algebra A reduces to a function sending elements of a finite set to elements of A,

```
Env : (A : Algebra {a} {ℓ₁}) → ℕ → Set _
Env A n = Fin n → ∥ A ∥
```

Furthermore, by the universal property of free algebras, given any environment θ, there is a unique homomorphism inst θ,

```
inst : ∀ {n} (A : Algebra {a} {ℓ₁}) → Env A n → F n ⇝ A
```

instantiating terms in F n in the algebra over which θ is defined.

4.2.6 Quotient Algebras

Working with setoids, quotient algebras can be defined straightforwardly. Given a setoid S with carrier A, the quotient of S by an equivalence _≈_ is the setoid induced by A under _≈_. Note, however, a quotient in this sense will only induce the inclusion we require when _≈_ is *coarser* than S's book equality (i.e., S's book equality is contained in _≈_).

If S is the carrier of an algebra, we can take the quotient of this algebra by _≈_ so long as this algebra's interpretation is a congruence over _≈_. In particular, the resulting algebra has carrier S / _≈_ but retains the original algebra's interpretation.

My implementation encodes these ideas directly alongside a proof that this construction satisfies the usual universal property of quotient objects.

4.3 Equational Theories

This section builds on the definitions given in the previous section, in order to support the specification of equational theories. Again, many of the core definitions are mono-sorted specialisations of those given by Gunther *et al.*, further dropping support for conditional equations.

4.3.1 Equations

An equation of Σ-algebras of arity n (i.e., containing at most n free variables) is defined as a pair of terms in the finitely generated algebra F Σ n,

```
Eq : (Σ : Signature) → (n : ℕ) → Set
Eq Σ n = ∥ F Σ n ∥ × ∥ F Σ n ∥
```

As examples, making use of syntax extensions to aid readability, associativity and commutativity of a binary operator can be encoded as follows,

```
comm : ops Σ 2 → Eq Σ 2
comm • = ⟨ a ⟩ ⟨ • ⟩₂ ⟨ b ⟩ , ⟨ b ⟩ ⟨ • ⟩₂ ⟨ a ⟩

assoc : ops Σ 2 → Eq Σ 3
assoc • =
  (⟨ a ⟩ ⟨ • ⟩₂ ⟨ b ⟩) ⟨ • ⟩₂ ⟨ c ⟩ , ⟨ a ⟩ ⟨ • ⟩₂ (⟨ b ⟩ ⟨ • ⟩₂ ⟨ c ⟩)
```

4.3.2 Theories

An equational theory is comprised of an algebraic signature and a set of equations. However, to allow pattern matching on equations, I separate equation names from their presentations as pairs of terms,

```
record Theory : Set₁ where
  field
    Σ   : Signature
    eqs : ℕ → Set
    _⟦_⟧ₑ : ∀ {arity} → eqs arity → Eq Σ arity
```

For example, commutative semigroups satisfy two equations: commutativity (in two variables), and associativity (in three). CSemigroupEqs captures this inductively as an indexed family of equation symbols,

```
data CSemigroupEq : ℕ → Set where
  comm : CSemigroupEq 2
  assoc : CSemigroupEq 3
```

These symbols are given interpretations using the equations defined in §4.3.1,

```
csemigroup-eqs : ∀ {n} → CSemigroupEq n → Eq Σ-magma n
csemigroup-eqs comm = L.comm •
csemigroup-eqs assoc = L.assoc •
```

thus yielding a presentation of the theory of commutative semigroups,

```
Θ-csemigroup : Theory
Θ-csemigroup = record { Σ    = Σ-magma
                      ; eqs = CSemigroupEq
                      ; _⟦_⟧ₑ = csemigroup-eqs
                      }
```

4.3.3 Models

An algebra A partially satisfies an equation, in an environment θ, if the instantiations of the left and right-hand sides of this equation under θ are equal up to book equality,

```
_⊨⟨_⟩_ : ∀ {n} → (A : Algebra {a} {ℓ})
           → Env A n → Eq Σ n → Set ℓ
A ⊨⟨ θ ⟩ (lhs , rhs) =
  | inst A θ | lhs =[ A ] | inst A θ | rhs
```

A is said to satisfy an equation if it partially satisfies the equation in any environment,

```
_⊨_ : ∀ {n} → Algebra {a} {ℓ} → Eq Σ n → Set (a ⊔ ℓ)
_⊨_ S eq = ∀ θ → S ⊨⟨ θ ⟩ eq
```

Given an equational theory Θ with signature Σ, an algebra A *models* Θ if it satisfies all of Θ's equations,

```
Models : Algebra {a} {ℓ} → Set (a ⊔ ℓ)
Models S = ∀ {n} → (eq : eqs Θ n) → S ⊨ (Θ ⟦ eq ⟧ₑ)
```

Thus, a setoid S is a model of Θ whenever it forms a Σ-algebra which models Θ,

```
record IsModel : Set (a ⊔ ℓ) where
  field
    isAlgebra : IsAlgebra S
    models : Models (algebra S isAlgebra)
```

Again, it is convenient to bundle a setoid together with a proof that it forms a model of a given theory,

```
record Model : Set (suc a ⊔ suc ℓ) where
  field
    ||_||/≈ : Setoid a ℓ
    isModel : IsModel ||_||/≈
```

4.3.4 Free Models

Given a theory Θ and an algebra A matching its signature, the following inductive definition describes syntactic equivalence up to Θ when applied to A (without computation),

```
data _≋_ : || A || → || A || → Set (a ⊔ ℓ) where
  refl    : ∀ {x} → x ≋ x
  sym     : ∀ {x y} → x ≋ y → y ≋ x
  trans   : ∀ {x y z} → x ≋ y → y ≋ z → x ≋ z
  inherit : ∀ {x y} → x =[ A ] y → x ≋ y
  cong    : ∀ {n} → (f : ops (Σ Θ) n)
            → ∀ {xs ys} → Pointwise _≋_ xs ys
            → A ⟦ f ⟧ xs ≋ A ⟦ f ⟧ ys
  axiom : ∀ {n} → (eq : eqs Θ n) → (θ : Env A n)
            → | inst A θ | (proj₁ (Θ ⟦ eq ⟧ₑ))
              ≋ | inst A θ | (proj₂ (Θ ⟦ eq ⟧ₑ))
```

While A's book equality does not necessarily reflect this equivalence, we can 'force' A to model Θ synthetically by quotienting,

```
Synthetic : Model
Synthetic = record { ∥_∥/≈ = ∥ A ∥/ _≋_
                   ; isModel = isModel
                   }
```

Applied to finitely generated algebras, this quotient yields a synthetic construction of the finitely generated models of a theory,

```
J : ℕ → Model
J = Synthetic ∘ F
```

4.3.5 Coproducts

Coproducts of models are specified directly in terms of the following data,

```
record IsCoproduct : Setω where
  field
    inl : ∥ A ∥ₐ ⇝ ∥ A+B ∥ₐ
    inr : ∥ B ∥ₐ ⇝ ∥ A+B ∥ₐ

    _[_,_] : ∀ {d ℓ₄} (X : Model {d} {ℓ₄})
           → ∥ A ∥ₐ ⇝ ∥ X ∥ₐ
           → ∥ B ∥ₐ ⇝ ∥ X ∥ₐ
           → ∥ A+B ∥ₐ ⇝ ∥ X ∥ₐ
```

For the sake of brevity, this reproduction omits the fields $\mathsf{commute}_1$, $\mathsf{commute}_2$ and universal. However, these fields are present in the implementation, and enforce that _[_,_] commutes as it should and is the unique such map (up to pointwise equivalence).

4.4 Free Extensions

Using the construction of finitely generated models and the definition of coproducts given in the previous section, it is now possible to give a formal definition of finitary, mono-sorted free extensions.

An *extension* for a theory Θ is any function from models and naturals to models,

```
Extension : Setω
Extension = ∀ {a} {ℓ} → Model {a} {ℓ} → ℕ → Model {a} {a ⊔ ℓ}
```

An extension is *free* if it sends every model A and natural n to the coproduct of A with J n,

```
IsFreeExtension : Extension → Setω
IsFreeExtension _[_] =
  ∀ {a ℓ} (A : Model {a} {ℓ}) (n : ℕ) → IsCoproduct A (J n) (A [ n ])
```

Thus, a *free extension* is simply an extension bundled with a proof that it is free,

```
record FreeExtension : Setω where
  field
    _[_] : Extension
    _[_]-isFrex : IsFreeExtension _[_]
```

Given an arbitrary model A of a theory Θ, the quotient of the algebra Free (Atoms ‖ A ‖/≈ n) by semantic equivalence up to Θ, as defined in §4.3, very nearly yields a free extension, but fails to permit reductions of static sub-terms. Extending the inductive definition of semantic equivalence with the following constructor resolves this issue,

```
evaluate : ∀ {n xs} → (f : ops (Σ Θ) n)
           → term f (map |inl| xs) ≈ |inl| (A ⟦ f ⟧ xs)
```

allowing for the construction of synthetic free extensions as quotients. Thus, for any theory Θ, we have a free extension SynFrex Θ, the synthetic free extension, whose carrier is the setoid of partially static terms over A.

Given any two free extensions for a theory Θ, $\_[\_,\_]_1$ & $\_[\_,\_]_2$, a model A and a natural n, we have the following maps,

```
to : ‖ A [ n ]₂ ‖ₐ ⇝ ‖ A [ n ]₁ ‖ₐ
to = (A [ n ]₁) [ inl₁ , inr₁ ]₂

from : ‖ A [ n ]₁ ‖ₐ ⇝ ‖ A [ n ]₂ ‖ₐ
from = (A [ n ]₂) [ inl₂ , inr₂ ]₁
```

It is a standard result that these maps must be mutually inverse (up to book equality). Thus, we have an equivalence,

```
iso : ‖ A [ n ]₁ ‖ₐ ≃ ‖ A [ n ]₂ ‖ₐ
```

Given an arbitrary free extension X, instantiating this equivalence for X and the synthetic free extension yields maps,

```
norm = to X SynFrex A n
syn = from X SynFrex A n
```

with norm corresponding to reflection and syn corresponding to reification. Moreover, given an environment θ over an algebra A, there is always a reduction map from a synthetic free extension of A back into A,

$$\text{reduce} : (\theta : \text{Env} \parallel A \parallel_a n) \to \parallel A\ [\ n\ ]_s \parallel_a \rightsquigarrow \parallel A \parallel_a$$
$$\text{reduce}\ \theta = A\ [\ \text{id}\ ,\ \text{interp}\ A\ \theta\ ]_s$$

Piecing these terms together, we obtain a term corresponding to the proof synthesis technique described by chapter 3,

$$\begin{aligned}\text{frexify} : \forall\ &\{lhs\ rhs : \text{Term}\ (\text{BT} \parallel A \parallel n)\}\\ &\to |\ \text{norm}\ |\ lhs \approxeq |\ \text{norm}\ |\ rhs\\ &\to |\ \text{reduce}\ \theta\ |\ lhs \approx |\ \text{reduce}\ \theta\ |\ rhs\end{aligned}$$

In particular, given an arbitrary theory Θ, a model A and two terms over A containing at most n free variables, so long as the reflections of these terms are equivalent in some free extension of Θ, their reductions are equivalent in A for any environment.

4.5 Summary

To summarise, this chapter described a formalisation of free extensions for finitary, mono-sorted algebras, outlining key design decisions and their justifications. As discussed in §4.1, the category of MLTT types does not support the direct construction of general quotients and, as such, free and finitely generated models of arbitrary equational theories cannot be constructed without the use of an embedding – in this case, setoids.

However, using setoids, I successfully constructed a framework for the specification of free extensions and, although not discussed in this chapter, using this framework, I was able to implement and verify free extensions for semigroups and commutative semigroups. As the focus of this work is the meta-theory of free extensions, the details of these implementations are omitted. However, for the interested reader, their designs are described in detail in [20]. Furthermore, several examples using these extensions are exhibited in the coming chapter.

5 — Utilising Reflection

The previous chapter described the development of a formalisation of free extensions in Agda, ultimately yielding a term describing the proof synthesis technique outlined in §3.1 – namely, frexify. However, in isolation, frexify is too impractical to use, as it requires the abstract syntax of each side of an equation to be spelled out before it can be applied. For non-trivial expressions, providing this syntax by hand is prohibitively time consuming, and does nothing but directly duplicate the term structure already present in the original expressions. The aim of this chapter is to describe how these issues are overcome using Agda's proof reflection system. Specifically, this chapter discusses the implementation of a tactic, capable of synthesising calls to frexify without requiring the syntax of terms to be spelled out explicitly.

However, Agda's reflection system is limited in a variety of ways. For example, the type it uses to encode terms is not dependent in any way, and as such, tactic programming is highly error prone. Thus, despite the tactic consisting of no more than around 500 lines of Agda, it was by far the most difficult component of the project to implement. Moreover, this code conveys little about its behaviour, particularly as its types are so uninformative. Thus, rather than picking apart a collection of cryptic macro snippets, this chapter restricts its attention to providing a conceptual overview of the tactic (§5.1) and exhibiting some examples of it in use (§5.2), concluding with a brief summary (§5.3).

5.1 Conceptual Overview

The approach taken to synthesising calls to frexify is relatively straightforward: first, we are given the tactic's goal in the form of an ununified meta-variable, alongside a user-provided free extension and model. By querying the type-checker for the type of this goal, we learn the type the synthesised proof term must land in. Typically, this will start with a number of universal quantifiers, which are systematically removed to yield an equation.

With an equation in hand, we can extract the raw syntax of its left and right-hand sides. An open sub-term detection algorithm then scans over this syntax, building up the single set of open sub-terms of the equation. This algorithm considers a term to be open, whenever its type is equal to the carrier type of the user-provided model and it contains one or more non-canonical sub-terms (e.g., a variable). The cardinality of the set of open sub-terms determines the number of variables we freely extend by.

Once the set of open sub-terms has been identified, we can build the abstract syntax of the two sides of the equation. This is done in the obvious depth-first way, but we must make sure to send any open sub-term to a single dynamic atom, and given multiple occurrences of the same open sub-term, ensure it is sent the same dynamic atom each time. The key remaining question is how operators are identified. However, by inspecting the user-provided model's signature, we can grab the constructors of its set of operator symbols. Taking the image of these symbols under the model's interpretation and normalising, we obtain syntactic representations of the definitions underlying the model's interpretation. A simple parser can then scan each side of the equation recursively, looking for fragments matching the definition of an operator, thus identifying the abstract syntax.

Once the syntax for the left and right-hand sides has been deduced, we convert the set of open sub-terms into an environment, and apply frexify to the two terms and this environment, assuming that the reflection of the two terms into the free extension will yield judgementally equal representatives. We wrap the resulting call to frexify with binders matching the quantifiers removed when extracting the original equation, and then unify the result with the goal meta-variable.

5.2 Examples

With an overview of the tactic given, we now turn our attention to some examples. First, returning to the example used throughout chapters 2 & 3, the tactic, fragment, can be instantiated with a free extension for commutative semigroups to synthesise a proof,

```
frexified : ∀ {x y} → (2 + x) + (y + 3) ≡ x + (y + 5)
frexified = fragment CSemigroupFrex +-csemigroup
```

Taking things a little further, in the following example, the tactic is required to correctly identify the applications $f\ 3$, $f\ x$ and $f\ (f\ 2)$ as open sub-terms and send each to a single dynamic atom, as discussed in §5.1,

```
symbols : ∀ {f : ℕ → ℕ} {x y}
            → (2 + f x) + (y + f 3) + f (f 2) ≡ (1 + f x) + (1 + y) + (f 3 + f (f 2))
symbols = fragment CSemigroupFrex +-csemigroup
```

The tactic is well-behaved in any context in which it can infer the left and right-hand sides of its goal. In particular, this means that the tactic can be dropped into reasoning-style proofs naturally,

```
+-inner : ∀ {m n k} → k * (m + 2) + k * (3 + n) ≡ k * (m + 5 + n)
+-inner {m} {n} {k} = begin
```

```
  k * (m + 2) + k * (3 + n)
≡⟨ sym (*-distribˡ-+ k (m + 2) (3 + n)) ⟩
  k * ((m + 2) + (3 + n))
≡⟨ cong (k *_) (fragment SemigroupFrex +-semigroup) ⟩
  k * (m + 5 + n)
∎
```

However, this example also demonstrates the tactic's extensibility, as in this instance, a free extension for semigroups is used in place of the free extension for commutative semigroups used in the previous two examples.

These final few examples demonstrate the power of making the entire framework universe polymorphic as discussed in §4.1. In particular, the following four proofs witness bijections in Set, again using a free extension for semigroups,

```
×-assoc₁ : ∀ {A B C : Set} → (A × (B × C)) ↔ ((A × B) × C)
×-assoc₁ = fragment SemigroupFrex ×-semigroup

×-assoc₂ : ∀ {A B C : Set} → ((A × B) × (B × C)) ↔ (A × (B × B) × C)
×-assoc₂ = fragment SemigroupFrex ×-semigroup

⊎-assoc₁ : ∀ {A B C : Set} → (A ⊎ (B ⊎ C)) ↔ ((A ⊎ B) ⊎ C)
⊎-assoc₁ = fragment SemigroupFrex ⊎-semigroup

⊎-assoc₂ : ∀ {A B C : Set} → ((A ⊎ B) ⊎ (B ⊎ C)) ↔ (A ⊎ (B ⊎ B) ⊎ C)
⊎-assoc₂ = fragment SemigroupFrex ⊎-semigroup
```

Notice, too, that these examples demonstrate the applicability of this proof synthesis framework to arbitrary equivalences. In this scenario, as bijection (_↔_) forms an equivalence over Set, we can talk about equality of types up to bijection, which the framework handles easily. Moreover, as the entire framework is constructive, these examples actually compute bijections.

5.3 Summary

In summary, while conceptually simple, constructing the tactic was a serious challenge. However, with it, we are able to synthesise an enormous variety of proofs, using any free extension with minimal effort. The key point here is that this tactic entirely eliminates the need for piece-meal proof-reflection enhancements for algebraic solvers: this single tactic understands enough about algebra to extract syntax trees from terms in any equational theory covered by the framework.

6 — Conclusion

This dissertation described the design and implementation of a tactic for the Agda proof assistant, capable of synthesising proofs of algebraic identities. Built on a formalisation of a class of mathematical constructions known as free extensions, this tactic enjoys a number of desirable properties:

Simple The tactic is simple to use. In all contexts, use of the tactic reduces to identifying a free extension for a theory, and the model of that theory to which the tactic's goal pertains. All other information is inferred automatically from the inferred type of this goal.

Flexible Leveraging proof reflection, the tactic is able to operate in a variety of contexts, including reasoning environments. Moreover, as the tactic is built on a universe-polymorphic formalisation of universal algebra, it is able to discover algebraic identities in types inhabiting any universe at a finite level.

Complete By construction, the proof synthesis technique described in this work is complete, in the sense that given any two provably equivalent terms in some algebraic theory and an appropriate free extension, it is guaranteed to discover an equivalence.

General As a consequence of the choice of Agda configuration within which free extensions are specified, the tactic is able to operate in a wide variety of Agda configurations, including Agda's propositionally extensional and cubical variants.

Extensible By providing a general-purpose framework for the specification of finitary, mono-sorted free extensions, the library described in this work permits end-users to author custom free extensions, without altering the library's internals. Moreover, the design of the tactic guarantees compatibility with any such extension, allowing bespoke algebraic solvers to leverage proof reflection automatically.

6.1 Future Work

The developments described in this dissertation leave many interesting avenues for future work. First and foremost, implementing a broader collection of free extensions is an obvious next step. Yallop *et al.* have already described free extensions for a host of mono-sorted algebraic structures including monoids, groups, and rings[20]. Reconstructing these free extensions within the framework set out by this dissertation, not only yields powerful proof synthesis tools, but further enables the formal verification of these extensions, demonstrating the correctness of the algebraic optimisations made in [20].

From a proof synthesis perspective, there are still a number of opportunities to improve on the work described here. For example, closer integration with Agda's type-checker could allow for terms to be identified with elements of a free extension automatically, and therefore treated up to algebraic equivalence without the need to invoke a tactic explicitly. A more primitive form of this would be to retain the tactic, but make it aware of a collection free extensions, selecting from among them based on the type-classes of operators in its goal.

Finally, as illustrated by §3.3, free extensions generalise to a much broader class of algebras than those characterised by the framework implemented as part of this work. Thus, a generalisation of the algebraic foundations of this work would allow for the specification and verification of free extensions for more exotic equational systems, with examples including multi-sorted algebras, infinitary algebras and second-order algebras. Given the observations made in §3.3.1 regarding the possible connection between normalisation-by-evaluation techniques and free extensions for second-order algebras, research along these lines is likely to yield deep insights into both techniques.

Bibliography

[1] Guillaume Allais et al. "Frex: indexing modulo equations with free extensions (Extended Abstract)". In: *TyDe*. Aug. 2020.

[2] T. Altenkirch et al. "Normalization by evaluation for typed lambda calculus with coproducts". In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 2001, pp. 303–310. DOI: 10.1109/LICS.2001.932506.

[3] Gilles Barthe, Venanzio Capretta, and Olivier Pons. "Setoids in Type Theory". In: *J. Funct. Program.* 13.2 (Mar. 2003), pp. 261–293. ISSN: 0956-7968. DOI: 10.1017/S0956796802004501. URL: https://doi.org/10.1017/S0956796802004501.

[4] Edwin C. Brady. "IDRIS —: Systems Programming Meets Full Dependent Types". In: *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*. PLPV '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 43–54. ISBN: 9781450304870. DOI: 10.1145/1929529.1929536. URL: https://doi.org/10.1145/1929529.1929536.

[5] William DeMeo. *The Agda Universal Algebra Library, Part 1: Foundation*. 2021. arXiv: 2103.05581 [cs.LO].

[6] Martín Escardó. *Type Topology*. https://github.com/martinescardo/TypeTopology. 2010-21-∞.

[7] Marcelo Fiore. "Isomorphisms of Generic Recursive Polynomial Types". In: *SIGPLAN Not.* 39.1 (Jan. 2004), pp. 77–88. ISSN: 0362-1340. DOI: 10.1145/982962.964008. URL: https://doi.org/10.1145/982962.964008.

[8] Marcelo Fiore and Chung-Kil Hur. "On the Construction of Free Algebras for Equational Systems". In: *Theor. Comput. Sci.* 410.18 (Apr. 2009), pp. 1704–1729. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.12.052. URL: https://doi.org/10.1016/j.tcs.2008.12.052.

[9] Marcelo Fiore and Chung-Kil Hur. "Second-Order Equational Logic (Extended Abstract)". In: *Computer Science Logic*. Ed. by Anuj Dawar and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 320–335. ISBN: 978-3-642-15205-4.

[10] Marcelo P. Fiore and Ola Mahmoud. "Second-Order Algebraic Theories". In: *CoRR* abs/1308.5409 (2013). arXiv: 1308.5409. URL: http://arxiv.org/abs/1308.5409.

[11] Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. "Formalization of Universal Algebra in Agda". In: *Electronic Notes in Theoretical Computer Science* 338 (2018). The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017), pp. 147–166. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2018.10.010. URL: https://www.sciencedirect.com/science/article/pii/S1571066118300768.

[12] Martin Hofmann. *Extensional Constructs in Intensional Type Theory*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 1447112431.

[13] Jason Z. S. Hu and Jacques Carette. “Formalizing Category Theory in Agda”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 327–342. ISBN: 9781450382991. DOI: 10.1145/3437992.3439922. URL: https://doi.org/10.1145/3437992.3439922.

[14] P. Martin-Löf. “Constructive Mathematics and Computer Programming”. In: *Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*. London, United Kingdom: Prentice-Hall, Inc., 1985, pp. 167–184. ISBN: 0135614651.

[15] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984.

[16] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013. arXiv: 1308.0729 [math.LO]. URL: http://homotopytypetheory.org/book.

[17] The Agda Team. *Agda 2.6.0.1*. Version 2.6.0.1. 2019.

[18] The Coq Development Team. *The Coq Proof Assistant*. Version 8.13. Zenodo, Jan. 2021. DOI: 10.5281/zenodo.4501022. URL: https://doi.org/10.5281/zenodo.4501022.

[19] Freek Wiedijk. “Comparing Mathematical Provers”. In: *Mathematical Knowledge Management*. Ed. by Andrea Asperti, Bruno Buchberger, and James Harold Davenport. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 188–202. ISBN: 978-3-540-36469-6.

[20] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. “Partially-Static Data as Free Extension of Algebras”. In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236795. URL: https://doi.org/10.1145/3236795.