LEVERAGING INFORMATION CONTAINED IN THEORY PRESENTATION

LEVERAGING INFORMATION CONTAINED IN THEORY PRESENTATIONS

BY

YASMINE SHARODA, M.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2021) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Leveraging Information Contained in Theory Presentations
AUTHOR:	Yasmine Sharoda M.Sc.
SUPERVISOR:	Jacques Carette and William M. Farmer
NUMBER OF PAGES:	x, 199

Abstract

Building a large library of mathematical knowledge is a complex and labour-intensive task. By examining current libraries of mathematics, we see that the human effort put in building them is not entirely directed towards tasks that need human creativity. Instead, a non-trivial amount of work is spent on providing definitions that could have been mechanically derived.

In this work, we propose a generative approach to library building, so definitions that can be automatically derived are computed by a meta-program. We focus our attention on libraries of algebraic structures, like monoids, groups, and rings. These structures are highly inter-related and their commonalities have been well-studied in universal algebra. We use theory presentation combinators to build a library of algebraic structures. Definitions from universal algebra and programming languages meta-theory are used to derive library definitions of constructions, like homomorphisms and term languages, from algebraic theory presentations. The result is an interpreter that, given 227 theory expressions, builds a library of over 5000 definitions. This library is, then, exported to Agda and Lean.

To my family,

You are my greatest blessing

Acknowledgements

Praise and gratitude be to Allah, the most gracious and the most beneficent.

I would like to express my sincere thanks to my supervisors Dr. Jacques Carette and Dr. William Farmer for their continuous support to my learning journey. Your expertise and feedback were invaluable in shaping this research direction and throughout my studies. I learned from you a lot about how to do research and communicate it. Many thanks for your help navigating graduate school. I am very thankful to Dr. Wolfram Kahl and Dr. Ridah Khedri for being on my supervisory committee. Your feedback, over the years, on this work is very appreciated.

I am grateful to Dr. Michael Kohlhase, Dr. Florian Rabe, and members of the KWARC team for inviting me to their research group at FAU Erlangen. The collaboration between our teams has been a great source of inspiration for this work.

Without the support of my family and friends, I wouldn't have been able to pursue this degree. Thanks for all the love and support. Thank you for always believing in me and encouraging me to explore different areas. I am most grateful to Ahmad, my husband, best friend and mentor. Thank you for always being there for me. I am so proud of the team we make.

Contents

List of Figures

Chapter 1

Introduction

A large library of formalized, ready-to-use mathematics has long been the pursuit of mathematicians and computer scientists. The influential QED manifesto [Boyer *et al.*, 1994], released in 1994, envisioned a library in which all mathematics is formalized and rigorously checked. The QED manifesto believed in one-formalization-fits-all approach to building this library. Diversity in mathematical formalizations was a big obstacle towards realizing the library described by QED. There was not an agreement even on which foundation to use for formalizing all of mathematics [Kohlhase and Rabe, 2016]. Since then, mathematical knowledge management (MKM) has become an active area of research framing a new vision for a large math library. The universal digital math library (UDML), described in [Farmer, 2004], is a collection of heterogeneous, intercommunicating systems and building this library is described as a grand challenge facing MKM.

Despite the many efforts dedicated to building math libraries, a large universal library has not become a reality.[1] One reason is that developing and maintaining

[1]However, in 2020, the mathlib team is making serious inroads in that direction [Team, 2019] .

libraries of mathematics requires a lot of person-power. One would want to believe that this human effort is put into the creative work of formalizing new pieces of knowledge. By examining current libraries of theorem provers, we know this is not always the case. The algebraic hierarchy has been formalized various times in different libraries, sometimes even within the same system [Geuvers *et al.*, 2002; Garillot *et al.*, 2009; Spitters and van der Weegen, 2010; Pottier, 2019]. In every formalization, the library developers had to provide all the definitions of the structures in the hierarchy and related constructions such as homomorphisms. We want to add more automation to the process of building libraries. We identify some sources of redundancy that can be eliminated and use the theory of `Monoid` as our running example. `Monoid` is an algebraic structure, a member of the algebraic hierarchy, that describes algebras with a carrier set and an associative binary operation over that set that has an identity element.

Handwritten Boilerplate. `Monoid` is defined in [Jacobson, 1985] as

> A monoid is a triple $(M, p, 1)$ in which M is a non-vacuous set, p is an associative binary composition (or product) in M, and 1 is an element of M such that $p(1, a) = a = p(a, 1)$ for all $a \in M$

The definition of `Monoid` is followed by the definition of its homomorphism as

> If M and M' are monoids, then a map η of M into M' is called a homomorphism if
>
> $$\eta(ab) = \eta(a)\eta(b), \qquad \eta(1) = 1, \qquad a, b \in M$$

More monoid-related constructions are defined, like submonoid, and quotient monoids. The same constructions are defined for `Group` and `Ring`.

Formal systems[2] present algebraic structures using axiomatic theories. `Monoid` and its notion of homomorphism are presented axiomatically in a minimal (imaginary) computer language as follows:

```
theory Monoid {
  A : type
  e : A
  op : A → A → A
  lunit : {x : A} → op e x = x
  runit : {x : A} → op x e = x
  assoc : {x y z : A} →
      op x (op y z) = op (op x y) z
}
```

```
theory MonoidHom {
  M1, M2 : Monoid
  hom : M1.A → M2.A
  pres-e : hom (M1.e) = M2.e
  pres-op : (x y : M1.A) →
            hom (M1.op x y) =
            M2.op (hom x) (hom y) }
```

Let us now define `Group` and `Group` homomorphism within the same language:

[2]We use the term *formal systems* to refer to all computer systems with logical foundations, be it automatic theorem prover (ATP), interactive theorem prover (ITP), specification system, or others.

```
theory Group {
  A : type
  e : A
  op : A → A → A
  inv : A → A
  lunit_e : {x : A} → op e x = x
  runit_e : {x : A} → op x e = x
  linverse : {x : A} →
          op x (inv x) == e
  rinverse : {x : A} →
          op (inv x) x == e
  assoc : {x y z : A} →
          op x (op y z) = op (op x y) z }
```

```
theory GroupHom {
  G1, G2 : Group
  hom : G1.A → G2.A
  pres-e : hom (G1.e) = G2.e
  pres-op : (x y : G1.A) →
            hom (G1.op x y) =
            G2.op (hom x) (hom y)
  pres-inv : (x : G1.A) →
            hom (G1.inv x) =
            G2.inv (hom x)  }
```

Notice how the two definitions of homomorphisms are similar and depend uniformly on the details of the theory. This observation is not specific to `Monoid` and `Group`. Generally, the homomorphism of a theory Γ is a mapping between two instances (algebras) of Γ and has 3 components: 1) the two instances of the theory, 2) the function `hom` that maps the carriers of the 2 instances, and 3) a preservation axiom `pres-op` for each operation symbol `op`. The preservation axioms follow the pattern

$$\texttt{\{x}_1\texttt{ .. x}_n\texttt{ ∈ A}_1\texttt{\} → hom (op}_1\texttt{ x}_1\texttt{ .. x}_n\texttt{) = op}_2\texttt{ (hom x}_1\texttt{) .. (hom x}_n\texttt{)}$$

where $\texttt{A}_1$ is the carrier of the first instance and the domain of the `hom` function. $\texttt{op}_1$ and $\texttt{op}_2$ are the instances of the function symbol `op` residing in the first and the second instances, respectively.

This definition of homomorphism is given to us by universal algebra [Whitehead, 1898], which studies commonalities between algebraic structures and define their related constructions. It defines an algebra as [McKenzie *et al.*, 1987][3]:

> An algebra is an ordered pair $\langle A, F\rangle$ such that A is a nonempty set and $F = \langle F_i : i \in I\rangle$ where F_i is a finitary operation on A for each $i \in I$. A is called the universe of $\langle A, F\rangle$, F_i is referred to as a fundamental or basic operation of $\langle A, F\rangle$ for each $i \in I$, and I is called the index set of the set of operation symbols for $\langle A, F\rangle$.

Libraries formalizing the algebraic hierarchy would contain axiomatic theories describing algebras and their related constructions, like homomorphism, subalgebra, quotient algebra, term language, etc. For every one of those constructions, universal algebra provides a uniform definition in terms of the components of the theory. It gives us the meta theory and the abstractions that enables us to instantiate those definitions for every theory. This suggests that we can have a program generate those constructions from the individual theories, instead of having library developers provide them manually. As there are many algebraic structures in mathematics and computer science and many constructions for each of them, this automation can save significant human effort. In this work, we provide a framework for generating these constructions.

Variabilities in Theory Presentations. Universal algebra gives us the right abstractions to implement the generation framework, but we need to start with a choice of a theory presentation from which the constructions will be computed. We have

[3] An axiomatic theory that describes an algebra will also have a field in the ordered pair for the axioms describing its properties.

shown the definition of `Monoid` in an imaginary language. But formal systems have different ways to define `Monoid`. In Figure 1.1, we show the definitions of `Monoid` in 5 different language. The 5 definitions refer to the same mathematical concept, but they look different. Each one has all the components needed to describe a `Monoid`. Yet, they also reflect the design decisions taken by the library developers. For example, the Haskell and MMT definitions exposes the fact that `Monoid` in these libraries is defined as an extension of `Semigroup` . This forces users of the definition to deal with `Semigroup` theory even if their formalization does not need to. The two Coq definitions takes two extreme views to the bundling problem [Team, 2019; Al-hassy *et al.*, 2019; Spitters and Van der Weegen, 2011] by either having the carrier and all the function symbols as arguments (the first definition) or having all elements of the theory as declarations of a record type (the second definition). The formalization of the Algebraic hierarchy in the Agda standard library is based on setoids (sets equipped with an equivalence relation). Therefore, we find an extra field of the definition of `Monoid` corresponding to the equivalence relation `_ ≈ _`.

Having design decisions embedded into the library definitions is a big usability problem. Users won't be able to use them in their projects unless they employ the same decisions. Otherwise, they are forced to redefine them. That leads to many libraries formalizing the same knowledge, even in the same language. Coq has at least 4 different algebra libraries [Garillot *et al.*, 2009; Geuvers *et al.*, 2002; Spitters and van der Weegen, 2010; Pottier, 2019]. In [Garillot *et al.*, 2009], the authors acknowledge this situation saying:

> "In spite of this body of prior work, however, we have found it difficult to make practical use of the algebraic hierarchy in our project to formalize

Haskell

```
class Semigroup a =>
      Monoid a
 where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)
  mconcat :: [a] -> a
  mconcat =
   foldr mappend mempty
```

Lean

```
 class monoid (M : Type u)
  extends semigroup M,
          has_one M :=
   (one_mul : ∀ a : M,
              1 * a = a)
   (mul_one : ∀ a : M,
              a * 1 = a)
```

Coq

```
class Monoid {A : type}
 (dot : A → A → A)
 (one : A) : Prop := {
  dot_assoc :
   forall x y z : A,
   (dot x (dot y z)) =
   dot (dot x y) z
  unit_left : forall x,
   dot one x = x
  unit_right : forall x,
   dot x one = x
}
```

Alternative Definition:

```
Record monoid := {
 dom : Type;
 op : dom -> dom -> dom
  where "x * y" := op x y;
 id : dom where "1" := id;
 assoc : forall x y z,
  x * (y * z) = (x * y) * z;
 left_neutral : forall x,
  1 * x = x;
 right_neutal : forall x,
  x * 1 = x;
}
```

Agda

```
 record Monoid c ℓ :
    Set (suc (c ⊔ ℓ)) where
  infixl 7 _•_
  infix 4 _≈_
  field
   Carrier : Set c
   _≈_ : Rel Carrier ℓ
   _•_ : Op₂ Carrier
   isMonoid : IsMonoid _≈_ _•_ ε

 record IsMonoid (• : Op₂) (ε : A)
  : Set (a ⊔ ℓ) where
  field
   isSemigroup : IsSemigroup •
   identity : Identity ε

   open IsSemigroup isSemigroup public

   identityˡ : LeftIdentity ε •
   identityˡ = proj₁ identity
   identityʳ : Rightdentity ε •
   identityʳ = proj₂ identity
```

MMT

```
theory Monoid : ?NatDed =
 includes ?Semigroup
 unit : tm u # e
 unit_axiom : ⊢ ∀ [x] = x * e = x

theory Semigroup : ?NatDed =
 u : sort
 comp : tm u → tm u → tm u
  # 1 * 2 prec 40
 assoc : ⊢ ∀ [x, y, z]
  (x * y) * z = x * (y * z)
 assocLeftToRight :
  {x,y,z} ⊢ (x * y) * z
          = x * (y * z)
  = [x,y,z]
   allE (allE (allE assoc x) y) z
 assocRightToLeft :
  {x,y,z} ⊢  x * (y * z)
           = (x * y) * z
  = [x,y,z] sym assocLR
```

Figure 1.1: Representation of `Monoid` theory in different languages.

the Feit-Thompson Theorem in the Coq system."

We seek to use a generative approach to building libraries that would compute derivable information from a theory presentation. We want to abstract over design decisions, so our generated definitions become accessible to more platforms and user projects.

1.1 Research Problem

We want to enhance the process of library development. Instead of having library developers provide every piece of detail in the library, we want to employ a generative approach to the development. The library developers would be providing expressions describing the definitions to be included. Our generator would produce those definitions.

We believe a generative approach is possible because definitions within a library are written in formal languages which provide uniform syntax for expressing information and universal algebra provides the definitions of many constructions in terms of the components of the algebraic structure.

A generative approach would have the following benefits:

- Reduce the human effort put into producing standard knowledge by internalizing this knowledge in the generator.
- Enhance the library maintainability. Library developers write generative algorithms to create and manipulate definitions. Changing design decisions leads to changes in the generative algorithm.

```
newtype M a = M { runM :: R.ReaderT Ctx IO a }
  deriving (Functor, Applicative, Monad, MonadIO, R.MonadReader Ctx)
  deriving (Input Mode, Output Mode, State Mode) via (Logged "activeMode" Mode)
  deriving (Input  [Text], Output [Text], State [Text]) via (From "logHistory")
  deriving (Input Bool, Output Bool, State Bool) via (From "shouldLog")
  deriving (Input Screens, Output Screens, State Screens) via (Logged "screenList" Screens)
  deriving (Input [SubTiler], Output [SubTiler], State [SubTiler]) via (Logged "yankBuffer" [SubTiler])
  deriving (Input OldMouseButtons, Output OldMouseButtons, State OldMouseButtons) via (Logged "oldMouseButtons" OldMouseButtons)
  deriving (Input (M.Map Text Atom), Output (M.Map Text Atom), State (M.Map Text Atom)) via (Logged "atomNameCache" (M.Map Text Atom))
  deriving (Input (M.Map Atom [Int]), Output (M.Map Atom [Int]), State (M.Map Atom [Int])) via (Logged "atomValueCache" (M.Map Atom [Int]))
  deriving (Input [Window], Output [Window], State [Window]) via (Logged "stackCache" [Window])
  deriving (Input FocusedCache, Output FocusedCache, State FocusedCache) via (Logged "focusedWindow" FocusedCache)
  deriving (Input (M.Map SDL.Window XRect), Output (M.Map SDL.Window XRect), State (M.Map SDL.Window XRect)) via (Logged "borderLocations" (M.Map SDL.Window XRe
  deriving (Input (M.Map Window XRect), Output (M.Map Window XRect), State (M.Map Window XRect)) via (Logged "windowLocations" (M.Map Window XRect))
  deriving (Input (M.Map Window [ParentChild]), Output (M.Map Window [ParentChild]), State (M.Map Window [ParentChild])) via (Logged "windowChildren" (M.Map Win
  deriving (Input ShouldRedraw, Output ShouldRedraw, State ShouldRedraw) via (Logged "shouldRedraw" ShouldRedraw)
  deriving (Input Conf, Output Conf, State Conf) via (Logged "configuration" Conf)
  deriving (Input ActiveScreen, Output ActiveScreen, State ActiveScreen) via (Logged "activeScreen" ActiveScreen)
  deriving (Input OldTime, Output OldTime, State OldTime) via (Logged "lastTime" OldTime)
  deriving (Input Docks, Output Docks, State Docks) via (Logged "knownDocks" Docks)
  deriving (Input DockState, Output DockState, State DockState) via (Logged "dockState" DockState)
  deriving (Input KeyStatus, Output KeyStatus, State KeyStatus) via (Logged "keyStatus" KeyStatus)
  deriving (Input Tiler, Output Tiler, State Tiler) via FakeTiler M
  deriving (Input NewBorders) via FakeBorders M
  deriving (Input MouseButtons) via FakeMouseButtons M
  deriving (Input (Int32, Int32)) via FakePointer M
  deriving (Input [XineramaScreenInfo]) via FakeScreens M
  deriving (Input SubTiler, Output SubTiler, State SubTiler) via Coerce SubTiler M
  deriving (Input RootWindow) via (FromInput "rootWindow")
  deriving (Input Display) via (FromInput "display")
  deriving (Input Font.Font) via (FromInput "fontChoice")
  deriving (Input XCursor) via (FromInput "cursor")
  deriving (Log LogData) via (Logger M)
```

Figure 1.2: Example of deriving information in Haskell.
source: Xest Window Manager Project on github.

- Increase the usability of library definitions by reducing the amount of design decisions embedded into them.

We are inspired by the *deriving* mechanism in Haskell. When defining a new datatype, a Haskell user can ask for some utilities to be readily available for them to use on that type. The Haskell compiler would then generate these functions for the user. Some of these are basic, like equality and printer, but the community has gone as far as giving users the chance to define their own templates for deriving instances, knows as the *deriving-via* technique [Blöndal *et al.*, 2018]. A pretty impressive example of deriving information is shown in Figure 1.2. Also, the Lens library [Lens Library, 2020] in Haskell, uses Template Haskell [Sheard and Jones, 2002] for the same purpose.

In this work, we address the following research questions:

RQ1 Can the uniformity provided by universal algebra be captured by a meta program that generates parts of an algebra library?

RQ2 What are the preconditions for generating this new information?

RQ3 What design decisions can be abstracted away and which can be reintroduced after the generation of new constructs?

RQ4 How would this affect the activity of library building?

RQ5 Can these generative algorithms be extended beyond the structure captured by universal algebra?

1.2 Contributions

These are the principal contributions of the thesis:

- Highlight the redundancy in libraries formalizing the algebraic hierarchy (in Chapter 4).
- Build a library of over 200 theories describing the algebraic hierarchy, implemented using the combinators in [Carette *et al.*, 2019] (Chapter 7).
- Compile a list of structures that can be generated from theory presentations (Section 3.2).
- Generate some of these constructions in Tog, a small implementation of a dependently typed language, in the style of Agda, Coq and Lean (Chapter 9).
- Export this implementation to Agda and Lean, (Chapter 10).

1.3 Broader Context

The Tetrapod project [Carette *et al.*, 2020a] envisions a software system in which 5 aspects of doing mathematics are integrated. These 5 aspects are organization, inference, computation, narration, and concretization. The system will have a tetrapodal structure with knowledge organization in the center and each of the 4 modes making one of the legs of the tetrapod, as shown in Figure 1.3.

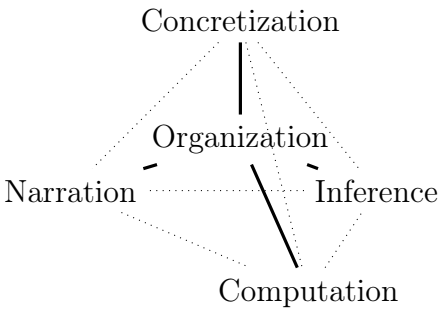


Figure 1.3: The tetrapodal structure of a mathematical software system that supports the five aspects of doing mathematics.

The organization aspect is reflected in the efforts of building libraries of mathematics. The Tetrapod project supports building a large library of mathematical knowledge organized as theory graph of biform theories [Carette *et al.*, 2018]. The theory graph structure connects theories by describing how the symbols of a source theory can be interpreted in the target one. In a graph, one can express facts like 'a group is a monoid' and that 'monoid and additive monoid are isomorphic'. We explain theories, morphisms and graphs in more details in Chapter 2. Ideally, we want the nodes of the theory graph to be biform theories [Carette *et al.*, 2018] which connect axiomatic theories (used by theorem provers) and algorithmic theories (used by computer algebra systems) using meaning formulas. This way communication

between reasoning and computation systems becomes possible. Communication can take the form of reasoning about algorithmic theories or using results of computer algebra systems in theorem provers.

This work contributes to the Tetrapod project by investigating how a generative approach can contribute to building the library at the center of the tetrapod. We focus on building the algebraic hierarchy and the constructions related to the theories in it, mainly as described by universal algebra. The library we build has a theory graph structure, but the nodes are axiomatic, rather than biform, theories.

1.4 Publications

The work on this thesis lead to the following publications; the following describe my role in these:

- [Carette *et al.*, 2018]

  Contributed to writing the project description of biform theories, mainly the motivation. The project description appeared in the proceedings of CICM 2018.

- [Carette *et al.*, 2019]

  Contributed to an extended paper discussing the MathScheme combinators that were initially published in [Carette and O'Connor, 2012]. The extended paper has been submitted to the *Journal of Automated Reasoning*. I contributed to surveying related work and framing the novelty of the work with respect to this related work, developing the type systems for

the combinators, and implementing them as discussed here in Section 8. I used this implementation to build a revised version of the MathScheme library.

- [Rabe and Sharoda, 2019]

  Used the diagram infrastructure developed in MMT [Rabe and Kohlhase, 2013a] and described in the paper to implement the MathScheme combinators described in [Carette *et al.*, 2019]. Since the combinators compute a theory and some arrows, we considered treating their inputs and outputs as diagrams. This was an earlier attempt to implement the combinators and also the first time diagrams combinators in MMT were tested. There were promising results, but they did not scale up since — at that time — there were problems with how MMT supports the diagram combinators.

- [Sharoda, 2019]

  Extended abstract submitted to the Doctoral Program at CICM 2019. The abstract was presented in the conference, but not refereed.

- [Carette *et al.*, 2020b]

  Presented redundancy in existing libraries and highlighted some of the problems we tackle in this thesis. Some of the main results of this thesis are published in the paper. I collected the examples of redundancy, and implemented and tested the framework.

- [Bercic *et al.*, 2020] (preprint)

Contributed to surveying and categorizing how different mathematics software organize knowledge. Knowledge organization is one of 5 categories of mathematics software the paper surveys. The other 4 are inference, computation, concretization, and narration.

1.5 Outline

We start by introducing some background knowledge in Chapter 2. We introduce universal algebra and the constructions of interest to this work in Chapter 3. We give in Chapter 4 examples of how some of these constructions are currently presented in libraries of formal systems, highlighting the redundancies that can be avoided. In Chapter 5, we introduce the methodology we use to enhance the library development process. We present Tog, the language and type checker that we use to develop our framework in Chapter 6. We discuss the combinators we use to build our library in Chapter 7, with the implementation discussed in Chapter 8. Chapter 9 discusses our generative framework that computes the constructions related to a specific theory. The theories and the generated constructions are exported to Agda and Lean. We discuss the exporter in Chapter 10. We present related work in Chapter 11. Conclusions and future work are discussed in Chapter 12.

Chapter 2

Background

Our ideas and implementation are based on dependent type theory (DTT). It is the meta theory for this work. We introduce it in Section 2.1 and define the notion of a theory and a context in Section 2.2.

Part of our work is building a library of axiomatic theories. The library is organized as a theory graph, in which theories are connected via morphisms. We introduce morphisms in Section 2.3 and discuss theory graphs and different strategies for building them in Section 2.4. To build the library we use combinators motivated by category theory, so we give a brief introduction for that in Section 2.5. Two of the constructions we generate are not typically defined within universal algebra texts. These are relational interpretations and staged terms, used in multi-staged programming. We give details on these in Sections 2.6 and 2.7, respectively.

2.1 Dependent Type Theory

Dependent type theory (DTT) is a version of type theory that enables writing types like Π `x : A · M x`, where the type `M x` depends on the *value* of `x`, i.e. `M : A → Type`. Having types that depend on values adds to the expressiveness of the logic. A common example for introducing dependent types is the type of a vector of `n` elements of a type `A`. In most programming languages, the type of this vector is defined in terms of the type of its elements as `Vec A`. Using dependent types, the type of a vector can depend on both the type of its elements and also its length, written as Π `n : ℕ · Vec A n`.

Having this extra information in the type allows detection of some errors, like accessing out of bounds elements, during type checking.

DTT is seen by many as a convenient foundation for representing mathematics [Gross *et al.*, 2020; Bauer, 2020; Shulman, 2010]. It lets one express ideas frequently used in mathematics. Statements like "the non-zero element", operations such as projecting the first unit vector in a specific dimension, or representation of a family of sets. The constructive nature of DTT adds the advantage that proofs can be run as programs.

Figures 2.1 and 2.2 shows the grammar and typing rules of a small version of dependent type theory with Π- and Σ-types like the one we use. The terms permissible in this type theory are variables, λ-abstractions, function applications, dependent type pairs, and their projections.

Σ-types. Types of dependent pairs, in which the type of the second element depends on value of the first one, are referred to as Σ-types. For example, Σ `n : ℕ · Vec A n`

t	::=		terms:
	|	$\mathbf{x}$	variable
	|	$\lambda(x : T).t$	abstraction
	|	$t\ t$	application
	|	(t,t)	typed pair
	|	$t.1$	first projection
	|	$t.2$	second projection
T	::=		types:
	|	$\mathbf{X}$	type/ family variable
	|	$\Pi\,\mathbf{x} : T.T$	dependent product type
	|	$T\ t$	type family application
	|	$\Sigma\,\mathbf{x} : T.T$	dependent sum type
K	::=		kinds:
	|	$*$	kind of proper types
	|	$\Pi\,\mathbf{x} : T.K$	kind of type families
Γ	::=		contexts:
	|	$\emptyset$	empty context
	|	$\Gamma, \mathbf{x} : T$	term variable binding
	|	$\Gamma, \mathbf{X} : K$	type variable binding

Figure 2.1: Grammar for a dependently typed language with dependent sum types. Adapted from [Aspinall and Hofmann, 2005].

$$\frac{x : T \in \Gamma \qquad \Gamma \vdash T : *}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash S : * \qquad \Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S \cdot t) : \Pi x : S \cdot T} \qquad \frac{\Gamma \vdash t_1 : \Pi x : S \cdot T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 t_2 : T[x \mapsto t_2]}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \equiv T' : *}{\Gamma \vdash t : T'} \qquad \frac{\Gamma \vdash \Sigma x : S \cdot T : * \qquad \Gamma \vdash t_1 : S \qquad \Gamma \vdash t_2 : T[x \mapsto t_1]}{\Gamma \vdash (t_1, t_2) : \Sigma x : S \cdot T}$$

$$\frac{\Gamma \vdash t : \Sigma x : S \cdot T}{\Gamma \vdash t.1 : S} \qquad \frac{\Gamma \vdash t : \Sigma x : S \cdot T}{\Gamma \vdash t.2 : T[x \mapsto t.1]}$$

Figure 2.2: Typing rules for a dependently typed language with dependent sum types. Adapted from [Aspinall and Hofmann, 2005].

refers to the type of a pair that contains the value of `n` $: \mathbb{N}$ in the first position and a vector of length `n` in the second one.

Telescopes. The concept of Σ types is generalized into that of *telescopes* or, equivalently, dependently-typed records [Pollack, 2002]. A telescope $\mathbb{T}$ is defined as

$$\mathbb{T} \equiv [\mathrm{x}_1 : \mathrm{A}_1][\mathrm{x}_2 : \mathrm{A}_2(\mathrm{x}_1)] \ ... \ [\mathrm{x}_k : \mathrm{A}_k(\mathrm{x}_1, ..., \mathrm{x}_{k-1})] \tag{2.1.1}$$

i.e. a sequence of typed name declarations where the type of later names can depend on earlier ones. The type `Vec A n` represented as a telescope would be written as `[A : Type][n :` $\mathbb{N}$`][Vec A n]`.

Contexts. In logic, a proposition is true if it is an axiom or is derivable from other true propositions using inference rules. This is usually written as $\varphi_1 ... \varphi_n \vdash \psi$. In categorical logic, instead of talking about propositions, one talks about contexts. [Pitts, 2001] defines a context as

> A context, Γ, is a finite list $[x_1 : \sigma_1, ..., x_n : \sigma_n]$ of (variable,sort) pairs, subject to the condition that $x_1, ..., x_n$ are distinct.

When using dependent types, the context becomes a telescope, where every type in the list can contain reference variables before it as described by Equation 2.1.1. The statement $\Gamma \vdash \psi$ means that the type judgement ψ follows from the context Γ. The concatenation of two contexts Γ_1 and Γ_2 is noted by $\Gamma_1; \Gamma_2$.

2.2 Theories

A theory Γ in some logic is defined as the tuple $(\mathcal{S}, \mathcal{F}, \mathcal{A})$ such that

- $\mathcal{S}$ is a set of sorts
- $\mathcal{F}$ is a set of function symbols.
- $\mathcal{A}$ is the set of formulas that hold in Γ.

The sorts in $\mathcal{S}$ and the function symbols in $\mathcal{F}$ constitute the language of the theory. The set $\mathcal{A}$ is closed under logical consequence and usually infinite. A *theory presentation* of a theory Γ includes a finite set of sorts, a finite set of function symbols, and a finite subset of $\mathcal{A}$ containing its generating axioms, i.e. axioms from which formulas that hold in Γ can be derived using inference. Note that the same theory can have different theory presentations. In this work, as is traditionally the case, we use the term theory to refer to *theory presentations*.

Theories as Contexts With dependent types and the Curry-Howard correspondence in place, the distinction between the three components of an axiomatic theory, sorts, function symbols, and axioms, is not needed anymore. Instead, a theory is seen as a Σ-type, dependently-typed context, or a telescope as described by Equation 2.1.1. For example, the axiomatic formalization of `Monoid` as a Σ type is:

```
Σ A : Type ·
 Σ op : A → A → A ·
   Σ e : A ·
     Σ lunit : {x : A} → op e x = x ·
       Σ runit : {x : A} → op x e = x ·
```

```
Σ assoc : {x y z : A} → op x (op y z) =
                        op (op x y) z
```

which can be describe as a telescope as:

```
Monoid = [A : Set, op : A → A → A, e : A,
          lunit : {x : A} → op e x = x,
          runit : {x : A} → op x e = x,
          assoc : {x y z : A} → op x (op y z) = op (op x y) z]
```

This definition induces a context from which the type `op e e = e` can be defined, which is noted as `Monoid ⊢ triv : op e e = e`

A theory presentation is well-typed if every declaration `c:t` is well-typed given its context. The formation rules for theory presentations are given in Figure 2.3, where $|\Gamma|$ refers to the list of symbols defined in the context Γ.

$$|\varnothing| = \varnothing \qquad |\Gamma\ ;\ x : \sigma| = |\Gamma| \cup \{x\}$$

$$\frac{}{\varnothing\ \texttt{ctx}} \qquad \frac{\Gamma\ \texttt{ctx} \qquad \sigma \notin |\Gamma| \qquad \Gamma \vdash \kappa : *}{(\Gamma\ ;\ \sigma : \kappa)\ \texttt{ctx}} \qquad \frac{\Gamma\ \texttt{ctx} \qquad x \notin |\Gamma| \qquad \Gamma \vdash \sigma : \kappa : *}{(\Gamma\ ;\ x : \sigma)\ \texttt{ctx}}$$

Figure 2.3: Formation rules for contexts as given in [Carette *et al.*, 2019]

2.3 Theory Morphisms

Morphisms are used to capture the structure of mathematics, by describing how theories are related to each other. In mathematical texts, a theorem proved for an

arbitrary `Monoid` can be used when considering an arbitrary `Group` without extra work. Formally, this can be done if a meaning preserving morphism between `Monoid` and `Group` exists. The morphism specifies how results in `Monoid` can be interpreted in `Group`.

A morphism $[v] : \Gamma \to \Delta$ consist of a list of assignments $[v]$, a source theory Γ, and a target theory Δ. $[v]$ assigns to every symbol[1] $x : \sigma$ in Γ a term $r : \sigma[v]$ in Δ. A term `t` in the language of Γ can be translated into a term `t'` in the language of Δ using substitution induced by the assignment $[v]$, such that `t'= t`$[v]$. Using the morphism `[op ↦ + ; e ↦ 0] : Monoid → AdditiveMonoid` we are able to interpret the expression `(op e x)` in `Monoid` as `(+ 0 x)` in `AdditiveMonoid` using substitution.

The formation rules for views are given in Figure 2.4.

$$\frac{\Delta\ \texttt{ctx}}{[] : \varnothing \to \Delta} \qquad \frac{(\Gamma\ ;\ x : \sigma)\ \texttt{ctx} \qquad [v] : \Gamma \to \Delta \qquad \Delta \vdash r : \sigma[v]}{[v, x \mapsto r] : (\Gamma\ ;\ x : \sigma) \to \Delta}$$

Figure 2.4: Formation rules for morphisms as given in [Carette *et al.*, 2019]

It is worth mentioning that the mapping is only a part of the morphism. A morphism consists of the source and destination theories as well as the mapping, i.e. the same substitution can induce different morphisms as the source and target are modified.

Connecting theories have been known for a long time in logic [Tarski *et al.*, 1953; Enderton, 1972] under the name *theory interpretations*. The same name is used by IMPS [Farmer *et al.*, 1993; Farmer, 1994]. Clear [Burstall and Goguen, 1980], OBJ, and their successors used the term *morphisms*, maybe because of using category

[1]The symbols of a theory are the names of its declarations.

theory for semantics. The term *view* has also been used to refer to the same concept by Maude, MathScheme, and MMT. In this work, we use the terms views and morphisms interchangeably.

We distinguish between three types of morphisms.

2.3.1 Identity Morphism

If $[v] : \Gamma \to \Delta$ is an identity morphism, then $[v]$ maps every symbol $x \in |\Gamma|$ to itself such that $x[v] = x$. While it is common to name source and target of identities with the same name, we do not do that here as Γ and Δ are two different theory presentations. The identity between them means that symbols in Γ are interpreted the same way in Δ.

Identity morphisms exist between two theories if the source is included verbatim in the destination, like in the case when describing a morphism from `Monoid` to `Group`. It is the simplest form of morphisms and allow transport of results without the need to perform substitution.

2.3.2 Embedding

If $[v] : \Gamma \to \Delta$ is an embedding, then $[v]$ maps every symbol $x \in |\Gamma|$ to a symbol $r \in |\Delta|$, which is not necessarily itself. $[v]$ is an injective mapping, and therefore is a bijection onto its range.

Consider for example, the following morphism from `Magma` to `AdditiveMagma`

$$\left\{ \begin{array}{lcl} A & : & \texttt{Type} \\ op & : & A \to A \to A \end{array} \right\} \xrightarrow{[A \mapsto A, op \mapsto +]} \left\{ \begin{array}{lcl} A & : & \texttt{Type} \\ + & : & A \to A \to A \end{array} \right\}$$

A term $t \in \Gamma$ is transported to Δ as $t[v]$, i.e.: by applying the substitution $[v]$ to the term t. So if $t = \ op\ x\ y$, where x and y are terms of type A, then using the morphism above it is transported to Δ as $(+\ x\ y)$.

We refer to an embedding morphism as $\widetilde{m}$, and therefore identity morphisms are referred to as $\widetilde{id}$.

2.3.3 General Morphism

A morphism in its general form is defined in the beginning of this section. An example is a morphism that flips a binary operation, i.e.: maps `op x y` to `op y x`

$$\left\{ \begin{array}{lcl} A & : & \texttt{Type} \\ op & : & A \to A \to A \end{array} \right\} \xrightarrow{[A \mapsto\ A, op \mapsto\ \mathsf{flip}\ op]} \left\{ \begin{array}{lcl} A & : & \texttt{Type} \\ op & : & A \to A \to A \end{array} \right\}$$

2.4 Theory Graph

One way to organize theories is using theory graphs. A theory graph is a directed graph consisting of theories as nodes and morphisms as edges between them. It is helpful in managing large libraries [Kohlhase *et al.*, 2010].

In systems that are based on categorical semantics, a theory graph is seen as a diagram in the category of theories and theory morphisms. Specware [Smith, 1999] uses the keyword *diagram* to build them. The work in [Autexier *et al.*, 2000], based on CASL, refer to them as *development graphs*.

Organizing a library as a theory graph leverages the structure of mathematics by relying on morphisms to connect the different concepts presented within the theories. Compare a library defining the graph leading to `Monoid` as in Figure 2.5 to one

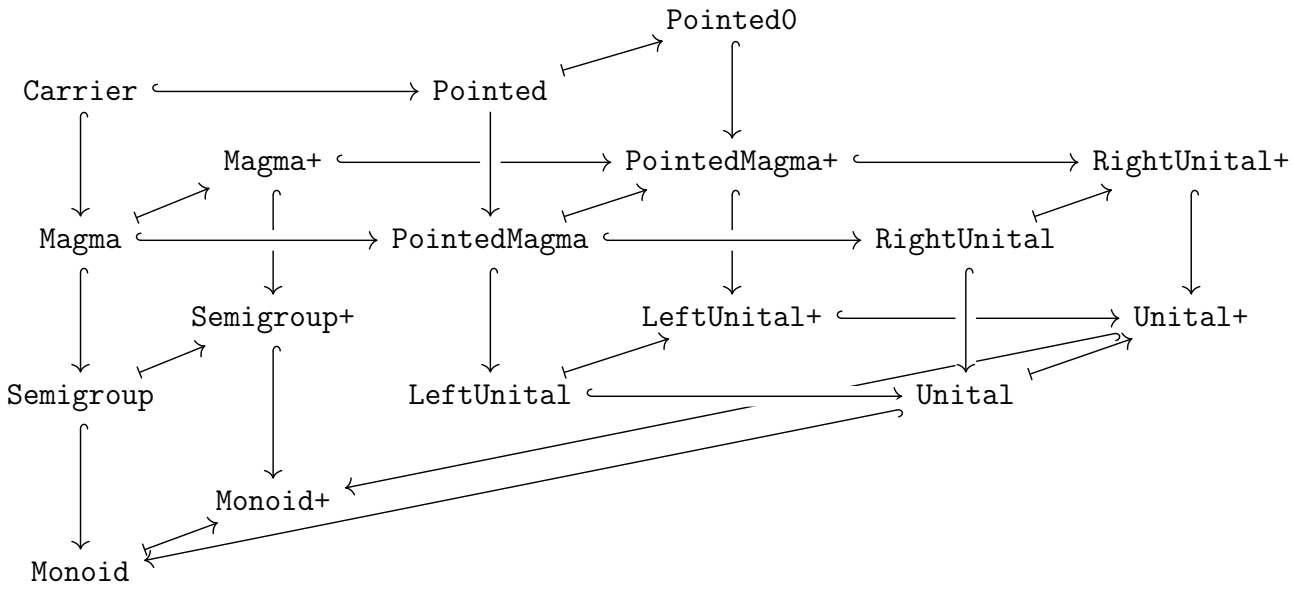


Figure 2.5: Structure of the algebraic hierarchy up to Monoids

that defines it only in terms of its components, as in Section 2.2. The theory graph provides more information which makes it more useful to library users. Theory graphs also make it possible to modularize a formalization by adding definitions or proving theorems within smaller modules (theories). Definitions and theorems are then made available to different other theories by transporting them via morphisms.

Here we discuss two strategies for decomposing theories; little and tiny theories.

2.4.1 Little Theories

The little theories approach is introduced in [Farmer *et al.*, 1992]. The idea is to ensure that if a statement `s` is proven in context Γ, then every statement in Γ is required to prove `s`. In this case, we say Γ is the *minimal axiomatization* needed to prove `s`. This implies that theorems are proved in different contexts based on the amount of structure needed to prove them. In contrast, the big theory approach

would use a small set of big theories for proving all results[2].

Using little theories increases the ability to reuse results. For example, if the theorem `op e e = e` is proven in the theory `Unital`, it can be transported to all theories that are connected to `Unital` via morphisms, like `Monoid`. On the other hand, if it is proven in the theory `Group`, it cannot be transported to `Monoid`, because all declarations in `Group` becomes part of the context for proving the theorem.

2.4.2 Tiny Theories

Tiny theories is a refinement of little theories. When building up a theory hierarchy in tiny theories style, only one new piece of information is added at a time [Carette *et al.*, 2011b]. To make this clear, let us consider a library that has the theories `PointedMagma` and `Unital` defined as follows.

```
theory PointedMagma = {
  A : Type
  e : A
  op : A → A → A }
```

```
theory Unital = {
  A : Type
  e : A
  op : A → A → A
  lunit : {x : A} → op e x = x
  runit : {x : A} → op x e = x  }
```

Defining `Unital` this way overlooks that in some cases one might want to define a theory to describe structures with a carrier and a binary operation on it that has only a right unit, like a theory with Integers as carrier and subtraction as the only binary operation. One will then need to add a new theory that is similar to `Unital` without the `lunit` declaration. Theorems proved in the context of `Unital` cannot be

[2]Or a medium-sized set of medium-sized theories

used, even if they only depends on `runit`.

Using tiny theories, one would first define a `LeftUnital` theory adding the `lunit` axiom to `PointedMagma`, a `RightUnital` theory adding `runit` axiom, and the theory of `Unital` would be connected to both `LeftUnital` and `RightUnital`, creating more connections and therefore, allowing more reuse of results. Systematically using tiny theories to develop a large library leads to the need for support to diamond structures, which we discuss in Chapter 7 based on the work in [Carette *et al.*, 2019].

2.5 Category Theory[3]

Category theory is a foundational framework, like set and type theory, that is abstract and structured enough to allow hidden commonalities of concepts to emerge.

While set theory has elements of sets as the main concept, category theory is built around the concept of morphisms. The source and target of a morphism are objects in the category. Category theory is not concerned with the internal structure of the objects, but rather by how they relate to other objects.

A category $\mathcal{C}$ consists of

- A collection of objects, $|\mathcal{C}|$
- For any two objects, a collection of morphisms between them. A morphism between objects Γ and Δ is presented as $u : \Gamma \to \Delta$.
- Operations assigning to every morphism its domain and codomain
- A composition function $\cdot$ assigning to each pair of morphisms $u : \Gamma \to \Delta$ and

[3]This section is based on [Pierce, 1990].

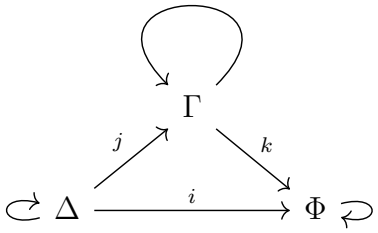


Figure 2.6: A diagrammatic representation of a category

$v : \Delta \to \Phi$, a morphism $v \cdot u : \Gamma \to \Phi$, such that for any arrow $w : \Phi \to \Omega$

$$w \cdot (v \cdot u) = (w \cdot v) \cdot u$$

i.e. $(\cdot)$ is associative.

- For every object Γ in $\mathcal{C}$, an identity morphism $id_\Gamma : \Gamma \to \Gamma$, such that for $u : \Gamma \to \Delta$

$$id_\Gamma \cdot u = u \cdot id_\Delta = u$$

i.e. id_Γ is a left unit for $(\cdot)$ and id_Δ is a right unit.

A diagram in a category $\mathcal{C}$ is a graph homomorphism between collection of vertices and directed edges (the shape of the diagram) to objects and morphisms of $\mathcal{C}$. Finite categories can be represented diagrammatically as in Figure 2.6. A diagram is said to commute if for every pair of vertices, Γ and Δ, all paths from Γ to Δ are equal, i.e.: compose to the same arrow.

In the following we introduce two concepts related to categories that we use in Chapter 7. These are pushouts and colimits. [nLab authors, 2020a] gives an intuition of what a colimit is as

> "The intuitive general idea of a colimit is that it defines an object obtained by sewing together the objects of the diagram, according to the instructions given by the morphisms of the diagram"

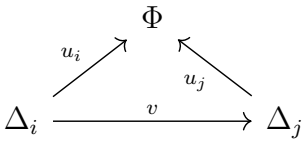
A pushout is a special case of a colimit. In [nLab authors, 2020b], it is mentioned that

> "A pushout is the colimit of the diagram $\bullet \longleftarrow \bullet \longrightarrow \bullet$ "

The formal definitions of the two constructions are given as follows:

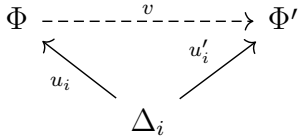
Colimits. Colimits are defined in terms of cocones. The definitions we present here are adapted from [Sannella and Tarlecki, 2012].

A cocone over a diagram D is an object Φ and a family of morphisms $u_0 : \Delta_0 \to \Phi, ..., u_n : \Delta_n \to \Phi$, where Δ_0 ... Δ_n are the objects in D, such that for every morphism $v : \Delta_i \to \Delta_j$ in D: $u_j \cdot v = u_i$, i.e. the following diagram commutes



. The notation used to describe cocones is $\langle u_i : \Phi \to \Delta_i \rangle_{i \leq n}$.

The colimit of a diagram is a cocone $\langle u_i : \Phi \to \Delta_i \rangle_{i \leq n}$ such that for any cocone $\langle u'_i : \Phi' \to \Delta_i \rangle_{i \leq n}$ there is a unique morphism $v : \Phi \to \Phi'$ such that for every u_i, the following diagram commutes



Pushouts. The pushout is the colimit of a diagram D that has exactly 3 objects and 2 morphisms. The morphisms need to have the same source. For a pair of morphisms $u_1 : \Gamma \to \Delta_1$ and $u_2 : \Gamma \to \Delta_2$, the pushout is an object Φ and a pair of morphisms $v_1 : \Delta_1 \to \Phi$ and $v_2 : \Delta_2 \to \Phi$ such that

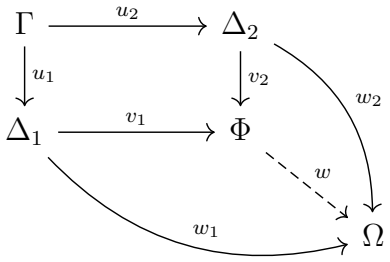


Figure 2.7: Diagram illustrating the definition of a pushout

- $v_1 \cdot u_1 = v_2 \cdot u_2$
- for morphisms $w_1 : \Delta_1 \to \Omega$ and $w_2 : \Delta_2 \to \Omega$, there is a unique $w : \Phi \to \Omega$, such that
  - $w \cdot v_1 = w_1$
  - $w \cdot v_2 = w_2$
  - $w_1 \cdot u_1 = w_2 \cdot u_2$

The definition of the pushout is illustrated in Figure 2.7.

2.6 Relational Interpretation

The relational interpretation of an algebraic structure is a generalization of its homomorphism [Reynolds, 1983] that defines a structure-preserving relation between carriers of its algebras. For example the relational interpretation between two monoid algebras $M_1 = (A_1, op_1, e_1, ...)$ and $M_2 = (A_2, op_2, e_2, ...)$ is a relation $interp : A_1 \to A_2$

such that:

$$interp\ e_1\ e_2$$

$$interp\ x_1\ x_2\ \wedge\ interp\ y_1\ y_2\ \rightarrow\ interp\ (op\ x_1\ y_1)\ (op\ x_2\ y_2)$$

Relational interpretations are mainly applied to models of type theories. They have been used in [Reynolds, 1983] to develop the abstraction theorem that connects meanings of expressions under different assignments and in [Plotkin and Abadi, 1993] to explain parameteric polymorphism. They have been applied to deduce theorems that apply to functions, given their polymorphic type [Wadler, 1989; Algehed *et al.*, 2020]. They are used extensively when working on the semantics of programming languages, often refered to as *logical relations* [Crary, 2005]. Supporting proofs of logical relations has been considered a benchmark for theorem provers in the revised POPLmark challenge [Abel *et al.*, 2019].

2.7 Multi-Stage Programming

Meta-programming is the practice of writing *meta* programs that manipulate *object* programs [Sheard, 2001; Sheard and Jones, 2002; Lilis and Savidis, 2019]. Meta and object programs can be in the same or different languages. Generative programming is one form of meta-programming in which the meta-program compiles into a program of the object language. Therefore, the process of running the meta-program involves at least two stages, compile and run-time.

The meta program might need to refer to code in the object language, like in the case of making a call to a predefined function in the object language. In this case, the

meta program is deferring the evaluation of this code to a `later` stage. Also, a meta program might need to evaluate a meta or object language expression that results in an object code. In this case, the expression is evaluated in the `current` stage.

In our implementation, we define two stages `s0` and `s1`.

```
data Stage : Set where
  s0 : Stage
  s1 : Stage
```

Staging an expression means adding annotation to its components indicating which stage it should be evaluated in, `Now` or `Later`.

```
data Staged (A : Set) : Set where
  Now : A -> Staged A
  Later : Comp A s1 -> Staged A
```

Annotating an expression of type `A` with the `Now` constructor indicates that it will be evaluated in the current stage and a value of type `A` is promised to exist. On the other hand, if the evaluation is deferred to `Later`, then the expression will have the type `Comp`, for computation.

```
data Comp (A : Set) (s : Stage) : Set where
  Computation : Choice -> CodeRep A s -> Comp A s
```

Computations encapsulate quoted fragments of code. The `CodeRep` function assigns a stage `s0` or `s1` to the expression.

```
data Wrap (A : Set) : Set where
  Q : A -> Wrap A
CodeRep : (A : Set) (s : Stage) -> Set
  CodeRep A s0 = A
  CodeRep A s1 = Wrap (CodeRep A s0)
```

We also add a flag indicating whether the quoted code represents an expression (`Expr`) or a literal, a constant or a variable (`Atom`).

```
data Choice : Set where
  Expr : Choice
  Atom : Choice
```

Staging has 3 main applications; generating well-typed code as in MetaOcaml [Taha, 1999], removing abstraction overhead introduced by generic programming [Yallop, 2016; Carette and Kiselyov, 2005; Carette *et al.*, 2011a], and developing domain specific languages [Sheard *et al.*, 2000]. MetaOcaml and Haskell templates provide staging constructs under the names *quote* and *eval* instead of *Now* and *Later*. In logical reasoning the same ideas are used for reflection, as in [Farmer, 2013].

Chapter 3

Universal Algebra: An Overview

Algebraic structures, like monoids, groups, and rings, are classes of algebras that have similar properties. Universal algbera studies those structures in a more generic way. It abstracts over the specific definitions and properties of classes of algebraic structures and deals with them as axiomatic theories in equational first-order logic. With this abstraction in place, universal algebra defines some constructions useful when dealing with algebras and prove some of their properties.

We use concepts of universal algebra to leverage the information in theory presentations. We internalize a representation of uni-sorted equational first order theories into DTT, our meta theory. This way we are able to manipulate them and generate the constructions as described by universal algebra. In this chapter we introduce core concepts that we use from universal algebra. In Chapter 9 we discuss how we use it in our work. In Section 3.1 we present equational first order logic, the meta theory for universal algebra, and define the components of a theory in this logic. We then introduce some of the constructions of universal algebra that can be generated from an equational theory presentation in Section 3.2. It is worth mentioning that

although our framework generates only some of these constructions, they all follow from the definition of a theory and the definitions we provide here will hopefully make this noticeable.

3.1 Equational Theory

Logics give us the machinery to describe properties of entities as formulas and reason about them. Equational logic restricts these formulas, whether axioms or theorems, to be universally quantified equations of the form $\mathtt{t}_1 = \mathtt{t}_2$, where $\mathtt{t}_1$ and $\mathtt{t}_2$ are terms expressible in the language of the theory. There are different notions of equality [Mazur, 2008; Grabowski *et al.*, 2015]. In many cases the underlying logic offers its own equality. In some other cases, the equality is defined by the language of the theory, as is the case with setoids.

Equational logic has 3 inference rules described in [Gries and Schneider, 1993]

$$\frac{t_1 = t_2}{t[x \mapsto t_1] = t[x \mapsto t_2]} \qquad \frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3} \qquad \frac{p\ t}{p\ (t[xs \mapsto ts])}$$

where t, t_1, t_2, and t_3 are expressions, x is a symbol in the language, ts is a list of expressions, xs is a list of symbols, and p is a predicate. The leftmost rule refers to Leibniz equality that states that two expressions are equal if one can be substituted by the other without changing the truth of a statement. The rule in the middle reflects the transitivity of equality. The rightmost rule states that if $p\ t$ is true, then it remains true under all substitutions.

A theory in universal algebra is described in first order equational logic. It restricts the definition of a theory described in Section 2.2. It is defined as a tuple $(\mathcal{S}, \mathcal{F}, \mathcal{E})$ such that

- $\mathcal{S}$ is a set of one sort $\mathtt{s}$.
- $\mathcal{F}$ is a finite set of function symbols along with their arities. A 0-ary function symbol is a constant.
- $\mathcal{E}$ is a finite set of generating equations.

An algebra ($\mathcal{S}$,$\mathcal{F}$) is a mathematical structure consisting of a domain and functions on this domain. It provides an interpretation for the carrier $\mathcal{S}$ and the function symbols in $\mathcal{F}$ of a theory.

3.2 Constructions

The definition of an equational theory captures various algebraic structures. To effectively use these structures, universal algebra provide us with definitions of constructions related to them. We will describe some of these constructions here. We use the symbol $\mathcal{S}$ to refer to the one sort in the set. We give the definitions of these constructs based on set theory, as one would find them in a standard text book. They have been formalized in type theory in both Coq in [Capretta, 1999; Spitters and van der Weegen, 2010] and agda [Gunther *et al.*, 2018]. The definitions are adapted from [Ehrig and Mahr, 1985] and [Meinke and Tucker, 1993].

- The *signature* of a theory $(\mathcal{S}, \mathcal{F}, \mathcal{E})$ is $(\mathcal{S}, \mathcal{F})$ consisting of the sort and n-ary function symbols, where $n \geq 0$. The signature specifies the language of the theory, without any laws.
- A *sub-theory* Δ of a theory Γ is a theory $(\mathcal{S}_\Delta, \mathcal{F}_\Delta, \mathcal{E}_\Delta)$ satisfying the conditions:
  1. $\mathcal{S}_\Delta \subseteq \mathcal{S}_\Gamma$

2. $\mathtt{c}_\Delta = \mathtt{c}_\Gamma \in \mathcal{S}_\Delta$ for every constant symbol in the set of function symbols $\mathcal{F}$.
3. $\mathtt{op}_\Delta\ \mathtt{x}_1 \ldots \mathtt{x}_n = \mathtt{op}_\Gamma\ \mathtt{x}_1 \ldots \mathtt{x}_n \in \mathcal{S}_\Delta$, for all $\mathtt{op} \in |\mathcal{F}|$, $\mathtt{x}_1 \ldots \mathtt{x}_n \in \mathcal{S}_\Delta$, and $\mathtt{n} \in \mathbb{N}$ such that $\mathtt{n} \geq 1$.

- The *trivial sub-theory* is the sub-theory with the empty carrier. Because the carrier is empty, the 3 conditions above trivially hold. Note that the trivial sub-theory is not defined for theories with constants.

- The *product* of two algebras $\mathtt{A}$ and $\mathtt{B}$ of the same theory Γ is a theory with sort $(\mathcal{S}_\mathtt{A} \times \mathcal{S}_\mathtt{B})$. If a theory is uni-sorted, then the set of sorts $\mathcal{S}$ is a singleton and we refer to that one sort as $\mathcal{S}$ for simplicity. In this case, the sort of the product theory is $(\mathcal{S} \times \mathcal{S})$.
  - $\mathtt{c}_\times : (\mathcal{S}_A \times \mathcal{S}_B) = c_\mathtt{A} \times c_\mathtt{B}$, for every constant symbol $\mathtt{c} \in |\Gamma|$.
  - $\mathtt{op}_\times : (\mathcal{S}_A \times \mathcal{S}_B) \rightarrow \ \ldots\ \rightarrow (\mathcal{S}_A \times \mathcal{S}_B)$, for every function symbol $\mathtt{op} \in |\Gamma|$ based on its arity, defined as:
    $\mathtt{op}_\times\ (\mathtt{x}_{1_\mathtt{A}}, \mathtt{x}_{1_\mathtt{B}}) \ldots (\mathtt{x}_{n_\mathtt{A}}, \mathtt{x}_{n_\mathtt{B}}) = (\mathtt{op}_\mathtt{A}\ \mathtt{x}_{1_\mathtt{A}} \ldots \mathtt{x}_{n_\mathtt{A}},\ \mathtt{op}_\mathtt{B}\ \mathtt{x}_{1_\mathtt{B}}\ \ldots \mathtt{x}_{n_\mathtt{B}})$
  - The set of equations $\mathcal{E}_\times$ is given by substituting the new sort, constant and function symbols in the equations in $\mathcal{E}$.

- A *homomorphism* between two algebras $\mathtt{A}$ and $\mathtt{B}$ of the same theory Γ is a function $\mathtt{hom} : \mathcal{S}_A \rightarrow \mathcal{S}_B$ such that
  - for every constant symbol $\mathtt{c}$ in $\mathcal{F}$: $\mathtt{hom}\ \mathtt{c}_A = \mathtt{c}_B$
  - for every function symbol $\mathtt{op}$ in $\mathcal{F}$:
    $$\mathtt{hom}\ (\mathtt{op}_A\ \mathtt{x}_1 \ldots \mathtt{x}_n) = \mathtt{op}_B\ (\mathtt{hom}\ \mathtt{x}_1) \ldots (\mathtt{hom}\ \mathtt{x}_n)$$

There are some variants of homomorphism that can be easily generated from it. These variants are

- *monomorphisms* are injective homomorphisms.
- *epimorphisms* are surjective homomorphisms.
- *endomorphisms* are homomorphisms from an object to itself.
- *isomorphisms* are bijective homomorphisms.
- *automorphisms* are isomorphisms from an object to itself.

- The *kernel* of a homomorphism from algebra `A` to algebra `B` of the same theory Γ is defined as the binary relation $\equiv_{hom}$ on the sort of `A`, such that

  $$\texttt{a} \equiv_{hom} \texttt{b} \Leftrightarrow \texttt{hom a} \equiv_{hom} \texttt{hom b}$$

  for every `a` and `b` in $\mathcal{S}_A$.

- The composition of two morphisms `f : A → B` and `g : B → C` is denoted by the function `g ∘ f : A → C` and is defined as `(g ∘ f) a = g (f a)` for every `a ∈ A`

- A *relational interpretation* between two algebras `A` and `B` of the same theory Γ is a relation `interp` $:\mathcal{S}_A \to \mathcal{S}_B \to \mathbb{B}$, such that

  - `interp` $\texttt{c}_A\ \texttt{c}_B$, where $\texttt{c}_A$ $\texttt{c}_B$ are the assignments of the constant `c` $\in \Gamma$ in algebras A and B, respectively.
  - `interp` $\texttt{x}_1\ \texttt{y}_1 \wedge \ldots \wedge$ `interp` $\texttt{x}_n\ \texttt{y}_n$
    $\Rightarrow$ `interp (op`$_A$ $\texttt{x}_1 \ldots \texttt{x}_n$`) (op`$_B$ $\texttt{y}_1 \ldots \texttt{y}_n$`)`,
    for all function symbols `op` $\in \mathcal{F}$, where $\texttt{x}_1 \ldots \texttt{x}_n \in \mathcal{S}_A$ and $\texttt{y}_1 \ldots \texttt{y}_n \in \mathcal{S}_B$.

- A *congruence relation* $\equiv$ for a theory Γ is an equivalence relation on elements of its sort which respects its operations, such that:
  - $\mathtt{x}_1 \equiv \mathtt{y}_1 \wedge \ldots \wedge \mathtt{x}_n \equiv \mathtt{y}_n \Rightarrow \mathtt{op}\ \mathtt{x}_1 \ldots \mathtt{x}_n \equiv \mathtt{op}\ \mathtt{y}_1 \ldots \mathtt{y}_n$
    for all function symbols $\mathtt{op} \in \mathcal{F}$.
- The *quotient algebra* for a theory Γ with respect to some congruence relation $\equiv$ is defined as the theory $\Gamma/\equiv = (\mathcal{S}_Q, \mathcal{F}_Q, \mathcal{E}_Q)$ such that
  - $\mathcal{S}_Q$ is the factor set of $\mathcal{S}$, defined as
    $$\mathcal{S}_Q = \{[\mathtt{x}] \mid \mathtt{x} \in \mathcal{S}\}$$
    where $[\mathtt{x}]$ is the equivalence class defined as $[\mathtt{x}] = \{\mathtt{y} \in \mathcal{S} \mid \mathtt{x} \equiv \mathtt{y}\}$
  - $\mathtt{c}_Q = [\mathtt{c}]$, for constant symbols $\mathtt{c} \in \mathcal{F}$ and $\mathtt{c}_Q \in \mathcal{F}_Q$.
  - $\mathtt{f}_Q[\mathtt{x}_1] \ldots [\mathtt{x}_n] = [\mathtt{f}\ \mathtt{x}_1 \ldots \mathtt{x}_n]$ for function symbols $\mathtt{f}_Q \in \mathcal{F}_Q$ and $\mathtt{f} \in \mathcal{F}$.

Term Languages

We define the term language of a theory, as well as some of its related constructions:

- The *closed term language* $\mathtt{L}$ induced by a theory is a set of terms that is defined inductively as
  - all constants belong to $\mathtt{L}$ (basic terms)
  - for every function symbol $\mathtt{op} : \mathcal{S} \to \ldots \to \mathcal{S}$ of arity n and for all terms $\mathtt{t}_1 \ldots \mathtt{t}_n \in \mathtt{L}$, the term $\mathtt{t}_{\mathtt{op}}\ \mathtt{t}_1 \ldots \mathtt{t}_n$.
- An *open term language* of a theory is similar to the closed term language, except that basic terms include the set of variables.

- The *staged term language* of a theory is the term language in which expressions can be marked for execution in compile or runtime stages as discussed in Section 2.7.

- *Induction Principle on Terms*: Let p be a predicate defined on terms $\mathtt{t} \in \mathtt{T_{op}(X)}$ of a signature $\mathtt{SIG} = (\mathcal{S}, \mathcal{F})$ with a set of variables X. The assertion p(t) is true for all $\mathtt{t} \in \mathtt{T_{op}}$ if the following conditions are satisfied:

  - (p t) is true for all constant and variable symbols t.
  - If $\mathtt{(p\ t_1)}, \ldots, \mathtt{(p\ t_n)}$ are true, then $\mathtt{p\ (f\ t_1 \ldots t_n)}$ is true, for every term $\mathtt{f\ t_1 \ldots t_n}$.

- *Evaluation functions*: Given an algebra A of a theory $\Gamma = (\mathcal{S}, \mathcal{F}, \mathcal{E})$, let T be the set of closed terms of the language of the theory as defined above; then the function $\mathtt{eval} : T \rightarrow \mathcal{S}_{\mathtt{A}}$ is defined recursively by:

  - $\mathtt{eval\ c} = \mathtt{c}_A$
  - $\mathtt{eval\ (op\ t_1 \ldots t_n)} = \mathtt{op}_A\ \mathtt{(eval\ t_1)} \ldots \mathtt{(eval\ t_n)}$

  The evaluation function for open term language would be similar except it has an additional environment that assigns value of the carrier to variables.

- *Simplification via rewriting*: Given a set of equations, each represented as (X,L,R), where X is a set of variables, L and R are terms of the language, L is the term on the left of the equation, and R is the term on the right side. By fixing the set of variables, we can represent equations as (L,R). Each equation represented in this form gives rise to two rewrite rules 1) $\mathtt{L} \Rightarrow \mathtt{R}$ and 2) $\mathtt{R} \Rightarrow \mathtt{L}$. Any of these can result in rewriting systems, but when simplifying

one need to define an ordering relation, which is a preorder (reflexive, transitive relation) that decides if a term is *simpler* than another. When having the equations and the ordering relation, a simplifier can be defined.

- *Equivalence of terms*: two terms can be denoted equal in one or more of the following cases
  - Evaluation of the two terms yields the same value.
  - Simplification of the two terms yield the same term.
  - The two terms are structurally identical, i.e.: they have the same syntax tree.

Chapter 4

Boilerplate in Libraries[1]

One of our observations is that current formalizations of algebra contain quite a bit of information that is "free" in the sense that it can be mechanically generated from basic definitions. For example, given a theory Γ, it is mechanical to define Γ-homomorphism.

Lest the reader think that our quest is a little quixotic, we first look at current libraries from a variety of systems, to find concrete examples of human-written code that could have been generated. We look at libraries in Agda and Lean in particular. More specifically, we look at version 1.4 of the Agda standard library and 2019 release of Lean's mathlib, where we link to the proper release tag.

We use the theory `Monoid` as our running example, and we highlight the reusable components that the systems use to make writing the definitions easier and more robust.

[1] This chapter is adapted from [Carette *et al.*, 2020b].

4.1 Agda Standard Library

The Agda standard library defines the following constructions related to `Monoid`:

1. Raw Monoid: The *raw* representation of a theory is a definition of its signature.. `RawMonoid` is defined in the standard library as

```
record RawMonoid c ℓ : Set (suc (c ⊔ ℓ)) where
  infixl 7 _•_
  infix 4 _≈_
  field
   Carrier : Set c
   _≈_ : Rel Carrier ℓ
   _•_ : Op₂ Carrier
   ε   : Carrier
```

   The definition of `RawMonoid` is identical to that of `Monoid` except for one declaration that instantiates the `isMonoid` record that checks for the properties of a `Monoid`.

2. Open Term Language and Evaluator: The "term language" of a theory is the (inductive) data type that represents the syntax of well-formed terms of that theory, along with an interpretation function from *expressions* to the carrier of the (implicitly single-sorted) given theory, i.e. its denotational semantics.

   In Agda, the definition of `Monoid` term language is straightforward:

```
data Expr (n : ℕ) where
  var : Fin n → Expr n
  id : Expr n
  _⊕_ : Expr n → Expr n → Expr n
```

Defining the interpretation function requires the concept of an environment. An environment associates a value to every variable, and the semantics associates a value (of type `Carrier`) to each expression of `Expr`.

```
Env : Set _
Env = λ n → Vec Carrier n


⟦_⟧ : ∀ {n} → Expr n → Env n → Carrier
⟦ var x ⟧ ρ = lookup ρ x
⟦ id ⟧ ρ = ϵ
⟦ e₁ ⊕ e₂ ⟧ ρ = ⟦ e₁ ⟧ ρ · ⟦ e₂ ⟧ ρ
```

These definitions are not found with the definitions of the algebraic structures themselves, but rather as part of the *Solver* for equations over that theory.

3. Product: Until recently, there was no definition of the product of algebraic structures in the Agda library. A recent pull request has suggested adding these, along with other constructions. The following hand-written definition has now been added:

```
monoid : Monoid a ℓ₁ → Monoid b ℓ₂ → Monoid (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
monoid M N = record
  { ε = M.ε , N.ε
  ; isMonoid = record
     { isSemigroup = Semigroup.isSemigroup
                       (semigroup M.semigroup N.semigroup)
     ; identity = (M.identityˡ , N.identityˡ <*>_)
                , (M.identityʳ , N.identityʳ <*>_)
     }
  } where module M = Monoid M; module N = Monoid N
```

where `semigroup` is the definition of the product theory of `Semigroup`.

4. Morphisms `Monoid` homomorphism is defined in the Agda standard library using `Magma` homomorphism as follows:

```
record IsMonoidHomomorphism (⟦_⟧: A → B) : Set(a ⊔ ℓ₁ ⊔ ℓ₂) where
 field
   isMagmaHomomorphism : IsMagmaHomomorphism ⟦_⟧
   ε-homo    : Homomorphic₀ ⟦_⟧ ε₁ ε₂
```

Monomorphism and isomorphism are also provided in the library, defined in terms of homomorphisms.

These constructions constitute 7 definitions spanning over 35 lines for only the theory `Monoid`. They are also repeated for other theories. The term language and evaluator for `Monoid` are repeated verbatim for both theories `CommutativeMonoid`

and IdempotentCommutativeMonoid. The Raw versions are provided for 7 theories; Magma, Monoid, NearSemiring, Semiring, Ring, and Lattice. The definitions of the 3 morphisms are provided for the same theories.

The direct product is defined for 10 theories. From the 7 that we defined above, only Magma, Monoid, and Group have definitions of direct product. In addition to those 3 theories, It is defined for Semigroup, Band, CommutativeSemigroup, Semilattice, CommutativeMonoid, IdempotentCommutativeMonoid, and AbelianGroup. Beside these definitions, the products of the signatures of Magma, Monoid, and Group is given in the library.

These give us a total of 47 definitions that are provided by the library developers, but could instead be generated, bearing in mind that not all constructions are provided for all theories. Also, constructions are not provided for additive or multiplicative versions of theories like Monoid and Group. A generative algorithm would be able to provide those variants of the constructions, at no extra cost.

It is worth noting that the defintions in the Agda standard library employ modularity when defining structures, like the definition of IsMonoidHomomorphism which depends on IsMagmaHomomorphism. Raw definitions from universal algebra do not support this modularity and, therefore, the generated expressions would be more *flat*, i.e. include the actual declarations instead of importing them from a different structure. Having flat definitions is, in some cases, a good way to abstract over library design. Nevertheless, we do not want to lose the connections between different theories. To solve this problem, we support a library organized as theory graph on which a flattener can be built. We leave working fully with unflattened theories as future work.

Summary

Construction	Number of Occurrences
Signatures	7
Homomorphisms	7
Monoomorphisms	7
Isomorphisms	7
Products	10
Products of Signatues	3
Term Language	3
Evaluation Function	3

4.2 Lean MathLib

The `homomorphism` of monoids is defined in two ways in mathlib. One way is the *unbundled* predicate style definition in which the homomorphism function is a parameter to the class definition.

```
class is_monoid_hom [monoid α] [monoid β] (f : α → β)
    extends is_mul_hom f : Prop :=
     (map_one : f 1 = 1)
```

where `is_mul_hom` is the definition of homomorphism of multiplicative magma, which lean refers to as `mul`. A very similar definition is provided for `add_monoid`. The other is the *bundled* definition in which the homomorphism function is part of the declarations of the structure, not a parameter to it.

```
structure monoid_hom (M : Type*) (N : Type*) [monoid M] [monoid N] :=
  (to_fun : M → N)
  (map_one' : to_fun 1 = 1)
  (map_mul' : ∀ x y, to_fun (x * y) = to_fun x * to_fun y)
```

The library provide the unbundled (class) definitions for many theories, including `group`, `semiring`, and `ring`. These definitions are marked deprecated. We were able to only find the bundled definitions for `monoid_hom`, its additive variant, and `ring_hom`.

The lean library also have definitions for the product of some theories. In a hierarchy ranging from `has_add` and `has_mul` to `nonzero_comm_ring`, 22 definitions of products are defined. It contains definitions of `is_submonoid`, `is_subgroup`, their additive variants, and `is_subring` .

Chapter 5

Methodology

We highlighted some of the problems that make library building labor intensive and suggest that by automating them we can lift some burden off the library developers. In this Chapter, we give more details on how we use automation for this purpose.

One of the main components of an algebra library is the axiomatic theory presentation of the algebraic structures, like the different formalizations of `Monoid` shown in Figure 1.1. In most theorem provers, developers provide all the declarations of the theory. Another way is to define theories by using combinators which describe how the new theory can be formed in terms of existing ones. Combinators are also a useful tool to leverage the structure of the theories by relating them to each other, which is useful when organizing the library as a theory graph. A *flattener* is used to compute the theory and morphisms resulting from the combinators. We discuss our implementation of the flattener in Chapter 8. Using these ideas, `Monoid` can be defined as

```
Monoid = combine Unital and Semigroup over Magma
```

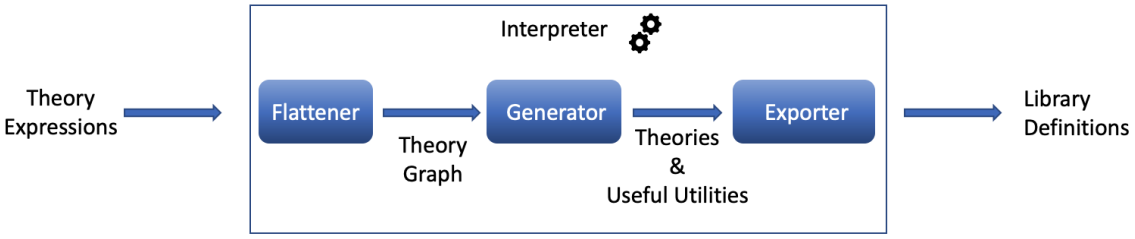


Figure 5.1: A 3-staged interpreter for generating libraries

Informally, this means that the theory of `Monoid` can be constructed by combining[1] the declarations in `Unital` and `Semigroup` without repeating the declarations in `Magma`.

The theories resulting from the flattener are used to compute some universal algebra constructions that are also part of algebra libraries. Chapter 4 shows examples of homomorphisms, product algebra and term languages provided by library developers. These and more can be generated based on their definitions from universal algebra. The *generator* does that by manipulating the components of the theories. We discuss the generator in Chapter 9.

The flattener and generator deal with mathematical definitions while keeping system-specific details to a minimum. In order to make the constructions more useful, the *exporter* makes them available in feature-rich systems, like Agda and Lean. We discuss the exporter in Chapter 10.

Figure 5.1 describes the 3-stage processing that a theory expression goes through. This process leads to the generation of all the constructions[2] described in Appendix B in Tog, Agda, and Lean.

[1]The `combine` operation is explained in detail in Section 7.2.3.

[2]The generated constructions can be found at: https://github.com/ysharoda/Deriving-Definitions/tree/115462d85389/Library/generated.

Chapter 6

Tog: Language and Type Checker

To implement the methodology we presented in Chapter 5, we need a language for representing and manipulating theories and a type checker to verify these manipulations. Theories are written in some formal language, the object language. To manipulate them we need to investigate and manipulate the syntax of the object language. This can be done in the same language if it has a strong reflection mechanism, or in the meta language in which the object language is embedded. As the main goal of our work is to investigate the usefulness of a generative approach, we do not want to be constrained by the amount of support given by the reflection mechanism. Working in the meta language gives us full control over manipulating the object language's syntax. We need our meta language to support the following features in the object language it represents:

- Π-Types: The semantics of the combinators we are using is given in categorical dependent logic. Having Π-types is needed to represent the types of views in terms of their source and target theories.

- Dependent records to represent theories as telescopes.
- A module system to manage namespaces such that every theory with its generated constructions is a module.
- Inductive data types to represent term languages.
- Equality to represent the equations within a theory.

These features are available in most dependently typed systems, like Agda, Coq, and Lean. But we refrained from using any of these systems to avoid delving into their design decisions. Instead, we prefer a small language that does not have many other extra features. We use Tog [Mazzoli *et al.*, 2017], a small implementation of Martin-Löf type theory. It provides a small dependently typed language and type checker. It was created by the Agda developers to experiment with type checking ideas. It has mainly been used to experiment with type checking through unification [Mazzoli and Abel, 2016].

Tog is implemented in Haskell. Figure 6.1 shows its internal representation.

A Tog module is a list of declarations, such that each declaration is either a type signature, function definition, datatype declaration, record definition, or a nested module represented using the `TypeSig`, `FunDef`, `Data`, `Record`, and `Module_` constructors, respectively. According to the type `Decl`, modules can import and open other modules, but our experience shows that this feature is not supported.

Parameters to modules, records, and datatypes are represented by the `Params` type. A single parameter has type `Binding` and can be declared implicit by using the constructor `HBind`.

```
data Decl
    = TypeSig TypeSig
    | FunDef Name [Pattern] FunDefBody
    | Data Name Params DataBody
    | Record Name Params RecordBody
    | Module_ Module
    | ...
    deriving (Eq, Ord, Show, Read)

data TypeSig = Sig Name Expr
    deriving (Eq, Ord, Show, Read)

data Where = Where [Decl] | NoWhere
    deriving (Eq, Ord, Show, Read)

data Params
    = NoParams | ParamDecl [Binding] | ParamDef [HiddenName]
    deriving (Eq, Ord, Show, Read)

data HiddenName = NotHidden Name | Hidden Name
    deriving (Eq, Ord, Show, Read)

data DataBody
    = DataDecl Name | DataDef [Constr] | DataDeclDef Name [Constr]
    deriving (Eq, Ord, Show, Read)

data RecordBody
    = RecordDecl Name
    | RecordDef Name Fields
    | RecordDeclDef Name Name Fields
    deriving (Eq, Ord, Show, Read)

data Fields = NoFields | Fields [Constr]
    deriving (Eq, Ord, Show, Read)

data Constr = Constr Name Expr
    deriving (Eq, Ord, Show, Read)

data FunDefBody = FunDefNoBody | FunDefBody Expr Where
    deriving (Eq, Ord, Show, Read)

data Telescope = Tel [Binding]
    deriving (Eq, Ord, Show, Read)

data Binding = Bind [Arg] Expr | HBind [Arg] Expr
    deriving (Eq, Ord, Show, Read)

data Expr
    = Lam [Name] Expr
    | Pi Telescope Expr     -- Π types
    | Fun Expr Expr         -- function types
    | Eq Expr Expr          -- equations
    | App [Arg]             -- type applications
    | Id QName              -- types names
    deriving (Eq, Ord, Show, Read)

data Arg = HArg Expr | Arg Expr
    deriving (Eq, Ord, Show, Read)

data Pattern
    = EmptyP Empty | ConP QName [Pattern] | IdP QName | HideP Pattern
    deriving (Eq, Ord, Show, Read)
```

Figure 6.1: Internal Representation of the Tog Language

A record field and a datatype constructor are both of type `Constr`, each having a name and a type expression `Expr`. Dependent types are created with the `Pi` constructor. Function types are curried and represented with the `Fun` constructor. Axioms that are equations are represented with `Eq` constructor. Type and function applications are created using the `App` constructor. The `Id` constructor is used for 0-ary types and functions, i.e.: If `q : QName`, then `q` is not a type, but `Id q` is.

To perform pattern matching, the `Pattern` type is used. Matching with a 0-ary constructor is done using `IdP`. If the constructor takes parameters, then `ConP` is used. `HideP` represents pattern matching on implicit arguments and `EmptyP` represents the don't care `_` character.

We extend Tog to support the input theory expressions and flatten them into Tog dependent records and morphisms between them. The structure of these dependent records is used to generate new constructions. The generated constructions can be records, datatypes, or functions presented in Tog syntax. The well-typedness of the generated constructions is ensured by the Tog type checker.

Chapter 7

A Library of Algebraic Structures

In this Chapter, we build a library of axiomatic theories representing the algebraic hierarchy. Our library consists of equational first-order theories organized as a theory graph using the tiny theories approach. Instead of having to provide all declaration of the theories and morphisms within the graph, we use the MathScheme combinators introduced in [Carette and O'Connor, 2012; Carette *et al.*, 2019].

It is common to see the algebraic hierarchy as a series of inclusions as in Figure 7.1. But the algebraic hierarchy is richer than that, considering for example the list in [Jipsen, 2019]. In Section 2.4.2 we discuss tiny theories as an adequate approach to building a theory graph that captures this structure. The nodes of the graph are theory presentations and they are connected via morphisms (see Section 2.4). Morphisms describe how the different theory presentations relate to each other. We present the example of building the theory of `Unital` by extending the theory of

$$\texttt{Magma} \hookrightarrow \texttt{Semigroup} \hookrightarrow \texttt{Monoid} \hookrightarrow \texttt{Group} \hookrightarrow \cdots$$

Figure 7.1: Algebraic structures as extensions.

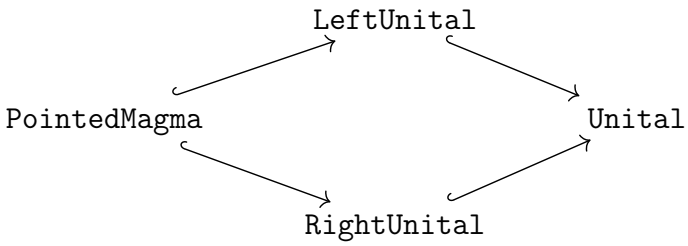


Figure 7.2: The diamond in the definition of `Unital`.

`PointedMagma` to create `LeftUnital` and `RightUnital`, then combining them. This example is described by a diamond structure as in Figure 7.2. The diamond structure appearing in the definition of `Unital` is not a special case. Instead, diamonds are pervasive in the algebraic hierarchy, as shown in the theory graph for defining `Monoid` in Figure 2.5.

But the diamond structure does not come without problems. We need to have careful infrastructure to deal with them in order to avoid the diamond problem [Bracha, 1992; Ducasse *et al.*, 2006; Wimmer *et al.*, 2011], a.k.a. multiple inheritance or the fork-join problem [Sakkinen, 1989], which we discuss in Section 7.2.3.

In Section 7.1 we provide an overview of the support for morphisms in different formal systems. Section 7.2 introduces the MathScheme combinators for a morphism-based approach to building theory graphs, leading to a solution to the diamond problem. We discuss how to use the combinators to build the library in Section 7.3. We end up with Section 7.4 discussing best practice for using the combinators.

7.1 Theory Graph Development

Although many formal systems support theory graph structures, more support for using and defining morphisms is needed. Specware [Smith, 1999] and MMT [Rabe

and Kohlhase, 2013b] force users to provide all details of theories and morphisms between them. IMPS [Farmer *et al.*, 1993], in some cases, generates morphisms given source and target theories.

Another way to support building a library rich in morphisms is to provide combinators to handle some of the work. Clear [Burstall and Goguen, 1980] is — to our knowledge — the first system to use combinators for creating new theories[1]. OBJ [Goguen *et al.*, 2000] and CASL [Mosses, 2004] are successors of Clear that also support combinators. We focus our discussion on CASL as a representative to these systems, as it is the only living one now and so we were only able to look at its library and run experiments on it. We realized two problems related to combinators in CASL. First, it is not always possible to flatten theories built through the use of combinators, especially hiding and freeness combinators [Mosses, 2004]. The second problem is related to how the *union* operation is implemented. The union operator is the one responsible for combining different specifications. They are combined on a 'same name, same thing' basis [Bidoit and Mosses, 2003], i.e. two declarations are considered the same if they have the same name. Figure 7.3 shows the problems that occur from using this principle. Both specifications `Ext1` and `Ext2`, on the left side, extend the `BaseSpec` with a binary operation and its unit element. A pushout between the two morphisms `BaseSpec` → `Ext1` and `BaseSpec` → `Ext2` would result in a theory with one sort, `A`, and two binary operations with two different unit elements. When trying this specification in CASL[2], it computes the declarations on the right side of the figure which has only one unit element for the two binary operations. This is different from what a pushout would compute.

[1] Clear is a specification language, and theories are used under the name specifications.
[2] Using the online tool at: http://rest.hets.eu

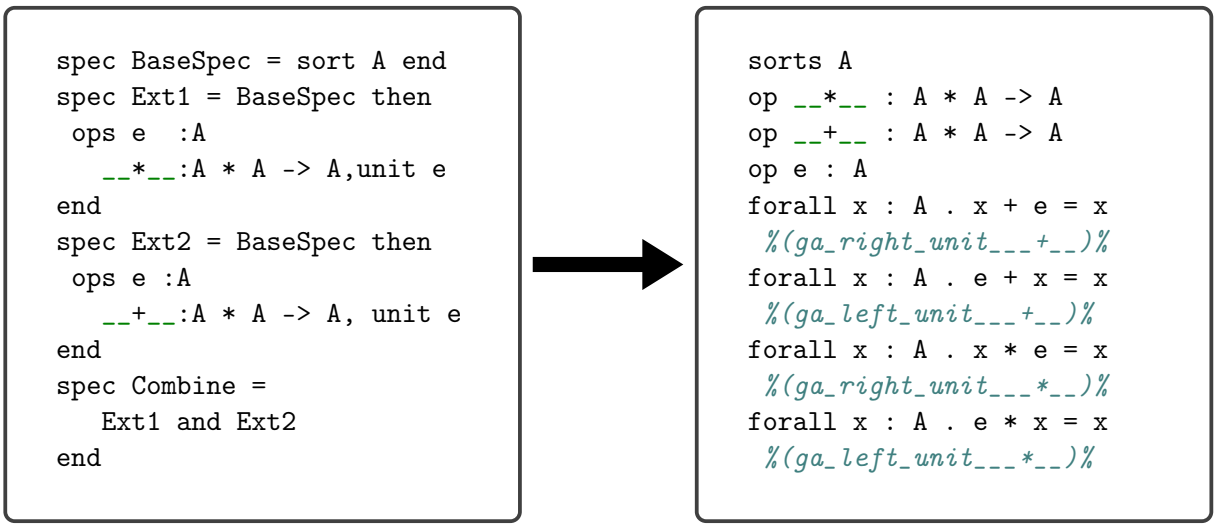


Figure 7.3: CASL union operation: On the left, the specification `Combine` is defined as the union of `Ext1` and `Ext2`. On the right, the declarations of specification `Combine` as computed by CASL.

We performed the same experiment with Isabelle locale expressions [Ballarin, 2003] and got similar results. In the following section, we introduce a collection of combinators that provide an infrastructure for building a large library organized as a theory graph that enables us to avoid the problem we have just described.

7.2 MathScheme Combinators

Combinators manipulate theories in different ways. They enhance modularity, reusability and maintainability of the library by saving the user the need to repeat definitions. [Carette *et al.*, 2019] introduces 4 combinators based on the definitions of theories as contexts and theory morphisms in dependent type theory as we discussed them in Sections 2.2 and 2.3.

A library built using these combinators embodies the following design decisions:

- Theories can always be flattened. Not all users of a formal system are interested in the hierarchy used to build the theories they need. A mathematician who wants to prove results in `Group` theory is only interested in groups with their standard definitions and results. This user should not be forced to work with groups as extensions of some theory, like `Monoid`. Abstracting over the hierarchy in users' code also has the advantage that the code need not change in case the hierarchy changes, like in the case of changing the type class hierarchy in Haskell [Haskell Wiki, 2015].

- Names are taken seriously. Similar concepts have different names in different contexts of mathematics. The unit of `_+_` has a different name than the one of `_*_` and confusing their names would be a huge usability problem. The combinators introduced in [Carette *et al.*, 2019] neither generate any names nor attempt to use any heuristics to solve name clashes. Instead name clashes are detected and the library developer is asked to resolve them.

- Tiny theories are systematically used. Since we do not provide a drop combinator, we use tiny theories to make sure all intermediate results are available for future theories to use.

- Morphisms are the main building unit of the library. The semantics and the implementation of the combinators are based on morphisms, not theories. This makes it possible to compute category-theoretic operations, like union, based on their real semantics, avoiding the need for assumptions like same-name-same-thing.

The combinators assume the underlying logic in which theories are defined to be

a dependent type theory (DTT). Therefore, a theory is viewed as a context, or a telescope as defined in equation 2.1.1. But a specific variant of DTT is not assumed; instead many of the details are abstracted away. The minimum requirements of the underlying DTT are listed in [Carette *et al.*, 2019]. We include them here for convenience and completeness. These requirements are:

- An infinite set $\mathbb{S}$ of symbols.
- A typing judgement for terms s of type σ in a context Γ which we write as $\Gamma \vdash s : \sigma$.
- A kinding judgement for types σ of kind κ in a context Γ which we write as $\Gamma \vdash \sigma : \kappa : *$. We further assume that the set of valid kinds $\kappa : *$ is given and fixed.
- A definitional equality (a.k.a. convertibility) judgement of terms s_1 of type σ_1 and s_2 of type σ_2 in a context Γ, which we write as $\Gamma \vdash s_1 : \sigma_1 \equiv s_2 : \sigma_2$. We will write $\Gamma \vdash s_1 \equiv s_2 : \sigma$ to denote $\Gamma \vdash s_1 : \sigma \equiv s_2 : \sigma$.
- A notion of substitution on terms. Given a list of symbol assignments $[x_i \mapsto s_i]_{i<n}$ such that they form a total function, and an expression e we write $e[x_i \mapsto s_i]_{i<n}$ for the term e after simultaneous substitution of symbols $\{x_i\}_{i<n}$ by the corresponding term in the assignment.

We now introduce the combinators we use from [Carette *et al.*, 2019].

7.2.1 Extension

Extension is the most basic combinator. On its own, it makes it possible to define a flat hierarchy as in Figure 7.1.

The inputs to an extension combinator are a theory presentation Γ and a list[3] of declarations $\Delta^+ = \{a_i : \sigma_i : \kappa_i\}_{i<n}$. The combinator computes a new theory $\Gamma \rtimes \Delta^+$ and an injective identity morphism ($\tilde{\mathtt{id}}$) from Γ to $\Gamma \rtimes \Delta^+$, where $\rtimes$ is an asymmetric operation that adds definitions to a telescope. On one side Γ is a well-formed theory, but Δ^+ may not be well-formed on its own. The construction is defined as:

$$\mathfrak{E}(\Gamma, \Delta^+) \triangleq \left\{ \begin{array}{l} \mathtt{pres} = \Gamma \rtimes \Delta^+ \\ \\ \mathtt{embed} = \tilde{\mathsf{id}} : \Gamma \to \Gamma \rtimes \Delta^+ \end{array} \right\}$$

where `pres` is the theory resulting from the extension and `embed` is the identity morphism from the theory being extended to `pres`.

An extension is well-formed if each new symbol $a_i : \sigma_i : \kappa_i$ in Δ^+ does not occur in Γ_{i-1} and its type is well-formed in Γ_{i-1}, where $\Gamma_{i-1} = \Gamma \rtimes \Delta_{i-1}$ and $\Delta_{i-1} \subseteq \Delta^+$ containing the first $i-1$ elements of Δ^+.

$$\begin{array}{l} \forall i \cdot a_i \notin |\Gamma_{i-1}| \\ \forall i \cdot \Gamma_{i-1} \vdash \sigma_i : \kappa_i \end{array}$$

where $\Gamma_{i-1} = \Gamma \rtimes \{a_0 : \sigma_0 : \kappa_0 \cdots a_{i-1} : \sigma_{i-1} : \kappa_{i-1}\}$.

Example Extensions are used when new concepts are added. According to little theories, the concept should be added in its smallest context, i.e. if $\Gamma \vdash c : t$ then for every $\Sigma \subset \Gamma$, $\Sigma \nvdash c : t$. Tiny theories encourages adding one new concept at a time. A good example is adding properties of a binary operation, like `commutativity` or

[3]As we use tiny theories approach, the list always has one declarations. The presentation here is more general and considers finite lists of any size.

`associativity` as follows[4]

```
Semigroup =
  extend Magma {assoc : {x y z : A} → op x (op y z) == op (op x y) z}
CommMagma =
  extend Magma {comm  : {x y : A} → op x y == op y x}
```

where `Magma` is the theory Γ being extended, `assoc` and `comm` are definitions in Δ^+.

7.2.2 Rename

A theory is a renaming of another if they contain the same declaration in the same order but with different names for the symbols. A useful use case for rename is obtaining boolean algebras from idempotent ring. Assuming some theorems have been proved for idempotent rings, these theorems still hold for boolean algebras and it would be useful to transport those theorems to boolean algebras without having to prove them again. This can be done if a rename morphism exists between the two theories. Renames allow using flexible notations while still reusing all results from the source theory.

Given a theory presentation Γ and a rename function π, the output of the rename operation is a new theory, `pres`, which is computed by performing a substitution of π into the declarations of Γ, and an embedding morphism $\tilde{\pi} : \Gamma \to \pi \cdot \Gamma$ that maps the

[4]The syntax we use here is the one used in our implementation. We give brief explanations for it here, and introduce it in details in the next section.

symbols of Γ to those of $\pi \cdot \Gamma$ based on the renaming function π.

$$\mathfrak{R}\,(\Gamma, \pi : |\Gamma| \to \mathbb{V}) \triangleq \left\{ \begin{array}{l} \texttt{pres} = \pi \cdot \Gamma \\ \\ \texttt{embed} = \tilde{\pi} : \Gamma \to \pi \cdot \Gamma \end{array} \right\}$$

A rename operation is well-formed whenever the rename function $\pi : |\Gamma| \to \mathbb{S}$ is an injection, and the codomain is a permutation of a subset of $\mathbb{S}$ with exactly k elements, where k is the number of declarations in Γ.

Example After defining `Semigroup` in the example of the previous section over a binary operation `op`, one would want to define the additive and multiplicative versions using the symbols `+` and `*`, resp. It also make sense to have a morphism from `Semigroup` to those variants that only differ in the names of the symbols. The rename combinator does just that:

```
AddSemigroup  = rename Semigroup {op to +}
MultSemigroup = rename Semigroup {op to *}
```

7.2.3 Combine

Conisder the following small library

```
Theory Empty = {}
Carrier = extend Empty {A : Set}
Pointed = extend Carrier {e : A}
Magma   = extend Carrier {op : A -> A -> A}
```

The flattened version of the theories of these libraries are

```
Empty = []

Carrier = [A : Set]

Magma = [A : Set, op : A → A → A]

Pointed = [A : Set, e : A]
```

Now we want to define the theory `PointedMagma` which has a binary operation and a point. It makes sense to assume this theory to be an extension of both `Magma` and `Pointed`. Using the extension combinator will not help us here. In this situation, we want a diamond in which our new theory is inheriting from two theories, but it is not clear whether a declaration, for example (`A : Set`) should be repeated or not. The situation is more complicated if we consider the definition of `AdditiveSemigroup` by relating it to `AdditiveMagma` defined as

```
AdditiveMagma = rename Magma {op to +}
```

and `Semigroup` defined as in Section 7.2.1. Here we have the same binary operation with different names. Which name should be used? Or should they be repeated, having two binary operations in the outcome?

The case when a theory needs to be related to more than one ancestor is prevalent when building large libraries. As we see in these examples, it occurs very early on when formalizing the algebraic hierarchy. The combine operation supports the multiple inheritance situation by relying on the information in the morphisms. Combine performs a pushout of the morphisms in the category of theory presentations, i.e. a pullback in the category of contexts. A pushout is a 5-ary operation that takes 2 morphisms and 3 objects of a category, as explained in Section 2.5. The morphisms

need to originate from the same source. The 3 theories can be deduced from the morphisms as the two target theories of the morphisms and their common source. For cases where there are name clashes, like the name clash between `op` and `+` in the `AdditiveSemigroup` example, the user is required to provide renames to resolve it. This is consistent with our design decision to not use heuristics or name generation to resolve any name conflicts.

The two morphisms of the combine operation u_Δ and u_Φ are both injective embeddings, having Γ as their source, and having Δ and Φ, resp, as their targets.

$$\mathfrak{C}\left(u_\Delta, u_\Phi, \pi_\Delta, \pi_\Phi\right) \triangleq \left\{ \begin{array}{l} \texttt{pres} = \Xi_0 \rtimes \left(\Xi_\Delta \cup \Xi_\Phi\right) \\ \texttt{embed}_\Delta = \left[v_\Delta\right] : \Delta \to \Xi \\ \texttt{embed}_\Phi = \left[v_\Phi\right] : \Phi \to \Xi \\ \texttt{diag} = \left[uv\right] : \Gamma \to \Xi \end{array} \right\}$$

π_Δ and π_Φ are two rename functions given to resolve name conflicts.
A well-formed combine needs to ensure that any two symbols $\texttt{x} \in |\,\Delta\,|$ and $\texttt{y} \in |\,\Phi\,|$ — after applying the renaming functions — map to the same symbol if they have originated from the source theory Γ and that there are no name clashes when mapping a symbol z across the two morphisms and rename functions. The precondition for combine operation is described by the following equivalence:

$$\pi_\Delta\left(x\right) = \pi_\Phi\left(y\right) \Leftrightarrow \exists z \in |\Gamma|\,.\; x = z[u_\Delta] \wedge y = z[u_\Phi] \tag{7.2.1}$$

Example We have given two examples in the beginning of this section illustrating situations in which combine operations is needed. A `PointedMagma` is defined as

```
PointedMagma = combine Magma {} Pointed {}
```

The embeddings being combined are `Carrier` ↪ `Magma` and `Carrier` ↪ `Pointed`. The empty `{}` means the identity rename functions are used in this expression, as in this case no name clashes need to be resolved.

The `AddSemigroup` is defined as

```
AdditiveSemigroup = combine AdditiveMagma {} Semigroup {op to +}
```

The embeddings used here are `Magma` ↦ `AdditiveMagma` and `Magma` ↪ `Semigroup`. The declaration `op` in `Magma` is mapped to `+` in `AdditiveMagma` and remains as `op` in `Semigroup`. Therefore, a rename `{op to +}` is needed to resolve this name clash.

7.3 Library Building

Using `extends`, `rename`, and `combine`, we build a library of 227 theories describing the algebraic hierarchy organized as a theory graph using tiny theories approach. Those theories range from `Empty` up to `Ring` and `BoundedDistributedLattice`. The library definitions are given in Appendix A. Our guide in building this library are the definitions in [Carette and O'Connor, 2011a], which were part of an experiment [Carette *et al.*, 2011b] on the way to developing the combinators we discuss in this chapter. Therefore, there are some definitions in that library that referred to non-existing morphisms, like the definition of `SemiRng` presented in Section 8. As the implementation of combinators depends on finding the right morphisms in the underlying theory graph, we had to work out the correct morphisms.

The examples in Section 7.2 give an intuition of how the combinators work together

to build the library. In this section we discuss some the challenges we faced to build the graph defining `AdditiveMonoid` as in Figure 2.5.

7.3.1 Defining AdditivePointedMagma

The very first theories of the algebraic hierarchy are defined as

```
Carrier = extend Empty {A : Set}

Pointed = extend Carrier {e : A}

Pointed0 = rename Pointed {e to 0}

Magma = extend Carrier {op : A -> A -> A}

AdditiveMagma = rename Magma {op to +}

PointedMagma = combine Pointed {} Magma {} over Carrier
```

These definitions would result in the black theories and morphisms in Figure 7.4. Now we want to defined `AdditivePointedMagma` consisting of three declarations `(A,+,0)` such that all the blue morphisms of Figure 7.4 are generated. Using one

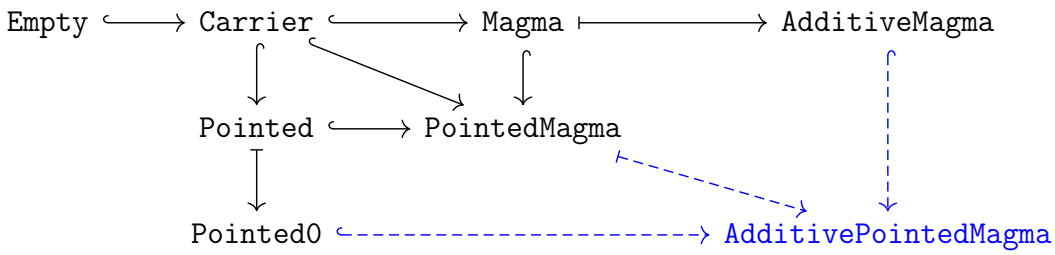


Figure 7.4: The construction of AdditivePointedMagma

`combine` to define it, we end up with the one of the following cases.

- `combine AdditiveMagma {} Pointed0 {} over Carrier`

would generate the theory `AdditivePointedMagma`, and the three morphisms

- `Carrier` → `AdditivePointedMagma`,
- `AdditiveMagma` → `AdditivePointedMagma`,
- `Pointed0` → `AdditivePointedMagma`.

The morphism `PointedMagma` → `AdditivePointedMagma` won't be generated.

- `combine AdditiveMagma {} PointedMagma {op to +} over Magma`

will not generate the morphism `Pointed0` → `AdditivePointedMagma`.

- `combine Pointed0 {} PointedMagma {e to 0} over Pointed`

will not generate the morphism `AdditiveMagma` → `AdditivePointedMagma`.

Instead, to get all these connections, we define `AdditivePointedMagma` as follows

```
Pointed0Magma =
  combine Pointed0 {} PointedMagma {e to 0} over Pointed
PointedPlusMagma =
  combine AdditiveMagma {} PointedMagma {op to +} over Magma
AdditivePointedMagma =
  combine Pointed0Magma {op to +} PointedPlusMagma {e to 0}
  over PointedMagma
```

which results in the graph in Figure 7.5. Although it is not immediately obvious to define `AdditivePointedMagma` this way, it corresponds more to the tiny theories

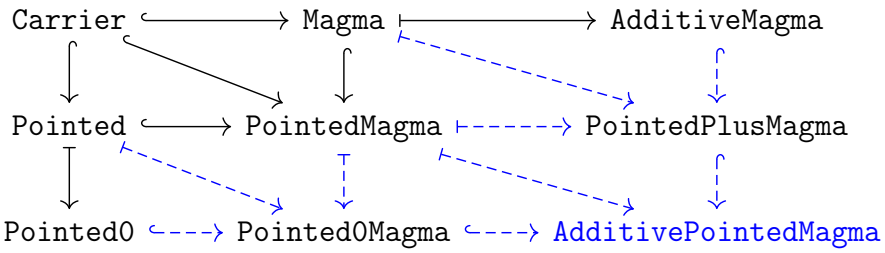


Figure 7.5: The construction of AdditivePointedMagma

approach that advocates having all intermediate theories. The two intermediate theories `Pointed0Magma` and `PointedPlusMagma` become useful when we define the `Zero0` theory, which is defined as follows:

```
PointedTimesZeroMagma =
  combine PointedTimesMagma {e to 0} Pointed0Magma {op to *}
  over PointedMagma
Zero0 =
  combine Zero {op to *; e to 0} PointedTimesZeroMagma {}
  over PointedMagma
```

7.3.2 Defining AdditiveMonoid

One would want to have `AdditiveMonoid` with all the morphisms we introduced in Figure 2.5. We have discussed the construction of `AdditivePointedMagma` and shown how the construction is not precisely depicted in Figure 2.5. Although all the morphisms are defined, some are composed of other morphisms. Now, we focus more on the part of defining `AdditiveUnital`.

The definition of `AdditiveLeftUnital` and `AdditiveRightUnital` goes as follows:

```
AdditiveLeftUnital =
  combine AdditivePointedMagma {} LeftUnital {op to +; e to 0}
  over PointedMagma
AdditiveRightUnital =
  combine AdditivePointedMagma {} RightUnital {op to +; e to 0}
  over PointedMagma
```

It make sense to expect `AdditiveUnital` to have morphisms with all of `Unital`, `AdditiveLeftUnital`, and `AdditiveRightUnital`. Similar to the case we had in the previous section, one pushout will only compute two of these three morphisms. The possible pushouts are

```
combine AdditiveLeftUnital {} AdditiveRightUnital {}
over AdditivePointedMagma

combine AdditiveLeftUnital {} Unital {op to +; e to 0}
over LeftUnital

combine AdditiveRightUnital {} Unital {op to +; e to 0}
over RightUnital
```

In order to compute the three, we need to to do 3 pushouts as follows:

```
AUnital1 = combine AdditiveLeftUnital {} Unital {op to +; e to 0}
           over LeftUnital
AUnital2 = combine AdditiveRightUnital {} Unital {op to +; e to 0}
           over RightUnital
AdditiveUnital = combine AUnital1 {} AUnital2 {} over Unital
```

The theories `AUnital1`, `AUnital2`, and `AdditiveUnital` are all equivalent. Therefore, the graph would have 3 presentations of the theory of additive unital without the graph realizing they are equivalent.

The same problem occurs when defining `AdditiveMonoid` and attempting to generate the three morphisms

- `AdditiveUnital` ⟶ `AdditiveMonoid`
- `AdditiveSemigroup` ⟶ `AdditiveMonoid`
- `Monoid` ⟶ `AdditiveMonoid`

We considered the possibility of using colimits or diagram combinators as in [Rabe and Sharoda, 2019]. In either case, we want to arrive at the right pushouts and build diagrams or colimits on top of that. Noticing that in all our experiments, the morphisms we wish to have that are not generated included an identity embedding, we adopted the solution of enabling the user to add those identity embeddings between theories.

The declarations that we use to define `AdditiveMonoid` are

```
Theory Empty = {}
Carrier = extend Empty {A : Set}
Pointed = extend Carrier {e : A}
Pointed0 = rename Pointed {e to 0}
Magma = extend Carrier {op : A -> A -> A}
AdditiveMagma = rename Magma {op to +}
Pointed0Magma =
  combine Pointed0 {} PointedMagma {e to 0} over Pointed
```

```
PointedPlusMagma =
  combine AdditiveMagma {} PointedMagma {op to +} over Magma
AdditivePointedMagma =
  combine Pointed0Magma {op to +} PointedPlusMagma {e to 0}
  over PointedMagma
Semigroup =
  extend Magma {assoc_op : {x y z : A} ->
                  op (op x y) z == op x (op y z)}
AdditiveSemigroup =
  combine AdditiveMagma {} Semigroup plus over Magma
LeftUnital = extend PointedMagma {lunit_e : {x : A} -> op e x == x}
RightUnital = extend PointedMagma {runit_e : {x : A} -> op x e == x}
AdditiveLeftUnital =
  combine AdditivePointedMagma {} LeftUnital {op to +; e to 0}
  over PointedMagma
AdditiveRightUnital =
  combine AdditivePointedMagma {} RightUnital {op to +; e to 0}
  over PointedMagma
Unital = combine LeftUnital {} RightUnital {} over PointedMagma
AdditiveUnital =
  combine AdditivePointedMagma {} Unital {op to +; e to 0}
  over PointedMagma
idUnital = id from AdditiveRightUnital to AdditiveUnital
Monoid = combine Unital {} Semigroup {} over Magma
```

```
AdditiveMonoid =
  combine AdditiveUnital {} Monoid {op to +; e to 0} over Unital
idMonoid = id from AdditiveSemigroup to AdditiveMonoid
```

Note that although we give names to identity morphisms, we never needed to refer to them in our development.

7.4 Discussion

In many cases, there are many ways to define a theory. We restrict using `extension` for adding new concepts within their minimal context, like adding associativity to `Magma`. Whenever associativity is needed in a different context, it should be transported through `rename` and `combine`. In other words, a concept should only be defined once and transported to different theories via morphisms. It is also reasonable to assume that `AdditiveMagma` should be an ancestor for any theory that contain the binary operation `+`. This means that many renames take place using `combine` operation, rather than the `rename` one. For example, compare the following two definitions of `AdditiveSemigroup`

1. `AdditiveSemigroup = rename Semigroup {op to +}`

2.
```
AdditiveSemigroup =
      combine AdditiveMagma {} Semigroup {op to +} over Magma
```

Definition 1 connects `AdditiveSemigroup` only to `Semigroup`, but definition 2 creates more embeddings and connects it to `AdditiveMagma`, `Semigroup` and `Magma`, which enriches the graph with useful morphisms.

We also find that using theories that are deeper in the hierarchy when possible adds more structure for the graph. For example, here are two possible definitions of `CommutativeGroup`:

```
1. CommutativeGroup =
       combine CommutativeMagma {} Group {} over Magma

2. CommutativeGroup =
       combine CommutativeMonoid {} Group {} over Monoid
```

The first definition does not connect `CommutativeMonoid` and `CommutativeGroup`, despite the fact that they are related. The second definition connects them, while also keeping the connection to `CommutativeMagma` through the path that exists from it to `CommutativeMonoid`.

These observations stem from the fact that we are not only interested in computing the output theory of the expression, but we are also interested in building a rich theory graph that captures as much of the structure of mathematics as possible.

In some cases, a whole hierarchy has been developed and one may want to perform a pushout of the whole graph along a morphism, in a similar way to [Rabe and Sharoda, 2019] and as shown in Figure 7.6. We encountered this situation while creating `Semiring`, as that is when the additive and multiplicative variants of the theories are combined together. We have not implemented diagram combinators in the Tog framework and leave this as future work.

Another line of future work is to support general morphisms as described in Section 2.3.3 and their usage in the `mixin` combinator as described in [Carette *et al.*, 2019]. The `mixin` combinator computes a pushout of an embedding along a general

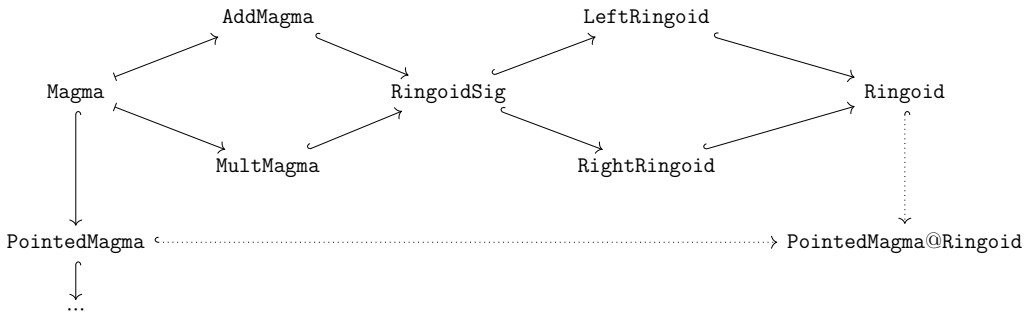


Figure 7.6: Shift the `PointedMagma` hierarchy to `Ringoid`

morphism. Given a general morphism $[\mathtt{u}_\Delta] : \Gamma \to \Delta$ and an embedding $[\mathtt{u}_\Phi] : \Gamma \to \Phi$ and two injective renaming functions $\pi_\Delta : |\,\Delta\,| \to \mathbb{S}$ and $\pi_\Phi : |\,\Phi\,| \to \mathbb{S}$, the mixin is defined as follows

$$\mathfrak{M}\left(u_\Delta, u_\Phi, \pi_\Delta, \pi_\Phi\right) \triangleq \left\{ \begin{array}{r} \mathtt{pres} = \Xi_1 \rtimes \Xi_2 \\ \mathtt{embed}_\Delta = [v_\Delta] : \Delta \to \Xi \\ \mathtt{view}_\Phi = [v_\Phi] : \Phi \to \Xi \\ \mathtt{diag} = [uv] : \Gamma \to \Xi \end{array} \right\}$$

where $\Xi_1 = \pi_\Delta \cdot \Delta$ is the theory presentation resulting from applying the rename function π_Δ to Δ via substitution. $\Xi_2 = \pi_\Phi \cdot \Phi^+$ is not a well-formed theory presentation, instead, it is the result of applying π_Φ to declarations of Φ that are not mappings of declarations in Γ. In [Carette *et al.*, 2019], a proof that the mixin operation as described above is always defined has been presented.

For example, the morphism `flip : Magma → FlippedMagma` shown in Section 2.3.3 can be used to construct flipped `Semigroup` as follows.

```
FlippedSemigroup = mixin flip {} Semigroup {}
```

In this case, u_Δ is the `flip` morphism, u_Φ is the morphism `Magma` $\rightarrow$ `Semigroup`. Therefore, the resulting presentation `pres` will have definitions from `FlippedMagma` and the associativity axiom from `Semigroup`.

Chapter 8

The Flattener

Theory expressions to theory graph

The combinators from [Carette *et al.*, 2019] has been implemented in [Rabe and Sharoda, 2019; Carette and O'Connor, 2011b; Al-hassy, 2019]. With the exception of [Rabe and Sharoda, 2019], the implementations and the associated libraries did not emphasize the morphisms in the way presented in [Carette *et al.*, 2019] and summarized in the previous chapter. Instead, different theories are combined using same-name-same-thing approach, which makes problems like the one in Figure 7.3 go undetected. This approach also computes results for expressions that should not be meaningful in the language of combinators presented in [Carette *et al.*, 2019]. Consider the following expression:

```
SemiRng = combine AdditiveCommutativeMonoid Semigroup Ringoid
          over RingoidSig
```

An implementation that reflects the principles of the combinators will not be

able to find a morphism between `RingoidSig` (the common source) and `Semigroup` (the second target) to compute the expression above. The theory `RingoidSig` has declarations for two binary operations, while `Semigroup` has only one. A morphism from `RingoidSig` to `Semigroup` needs to drop one binary operation. This is not possible given the choice of combinators that avoids a drop operation.

It is worth noting that by implementing the combinators we mean computing a flattened version of the theory presentation described by the given expressions. This is performed by the flattener that given a theory presentation produces a Tog dependent record of declarations within the described theory presentation.

In Section 8.1 we discuss a modification in the syntax of combine and why we have it. We introduce the syntax of the language we implement in Section 8.2 and start discussing the implementation in Section 8.3 by presenting how we represent theories and morphisms in our framework. In Section 8.3.2 we present the type of the theory graph. The implementation of the combinators that build the graph is presented in Section 8.3.3.

8.1 Referring to Morphisms

The extension and rename combinators need to identify a theory in the graph to operate on and compute the output theory and morphism. The input theory is part of the expression of the combinator. In the case of `combine`, the inputs to the combinator are two morphisms and two rename functions. But the syntax for combine is not defined in terms of morphisms. Instead, it is defined in terms of theories and the morphisms are left to the implementation to infer them. For example, the expression

```
combine CommutativeMagma {} AssociativeMagma {}
```

does give information that the targets of the two embeddings involved are the theories `CommutativeMagma` and `AssociativeMagma`, but it does not specify the source of the embeddings. The algorithm has three choices of the source theory, which is common to both morphisms.

- If the source theory is `Magma`, the theory resulting from the combine operation will have one binary operation that is both associative and commutative
- If the source theory is `Carrier`, then the definition is describing a theory (along with the related morphisms) that has two binary operations, one associative and the other commutative. But this theory will not be computed because of the name clash; The user has to choose another name for one of the two operations. A possible fix is

  ```
  combine CommutativeMagma {op to +} AssociativeMagma {op to *}
  ```

As the hierarchy gets deeper, this problem becomes more complicated. For example, `CommutativeGroup` and `IdempotentGroup` have many more possibilities for their common source.

The reason of this problem is that the syntax of the language presented in [Carette *et al.*, 2019] is based on naming target theories, while the syntax is based on having the embedding available. This leaves the gap of using the target theories to infer the embeddings. Using theories, instead of morphisms, in the syntax is a usability decision. Morphisms do not have canonical names, mainly because they do not appear

in informal mathematics. For example, it is hard to think of a name for the morphism (that result of composition of morphisms) from the `PointedMagma` to `Monoid` theory. It is easier to refer to it in terms of the source and the target than to give it any name.

We use an approach that still uses theories for usability reasons but gives more information for inferring the embeddings. We modify the syntax of `combine` in the paper to have an `over` part similar to the initial work on the combinators [Carette and O'Connor, 2012].

8.2 Theory Expressions

The language that we implement has the following syntax

```
Map m = {a_0 to b_0 ; ⋯ ; a_n to b_n}
Theory T = {a_0 : t_0 ⋯ a_n : t_n}
T′ = extend T {a_0 : t_0 ⋯ a_n : t_n}
T′ = rename T m
T′ = combine T_1 m_1 T_2 m_2 over T
i = id from T_1 to T_2
```

where `T`, `T′`, $\texttt{T}_1$, and $\texttt{T}_2$ are theories, `m`, $\texttt{m}_1$, and $\texttt{m}_2$ are mappings that can be either previously defined using the `Map` keyword or expanded as a list of mappings $\{\texttt{a}_0 \texttt{ to } \texttt{b}_0\texttt{; ... ; } \texttt{a}_n \texttt{ to } \texttt{b}_n\}$.

Although one can declare a theory with a list of declarations using the `Theory` keyword, we only use it to create the empty theory.

8.3 Implementation

In Chapter 7, we described the library we are building. This library is the output of the flattener. In Section 8.2, we introduce the theory expressions we need to generate this library. These expressions are the inputs to the flattener. In the sequel of this chapter, we describe the implementation of the flattener that reads those theory expressions and generates the graph.

The graph consists of theories and morphisms. In Section 8.3.1 we describe theories and morphisms, which are the basic components of the graph. We use them to define the theory graph data structure in Section 8.3.2, as well as the definition of a library. The combinators add the theories and morphisms to the graph. Their implementation is described in Section 8.3.3.

8.3.1 Theories and Morphisms

Theories are the building blocks of the library. We defined a theory in DTT in Section 2.2 as a telescope. It is captured by the type `GTheory`.

```
data GTheory = GTheory {
  declarations :: [Constr],
  waist        :: Int     }
```

The waist is needed to determine how many of the declarations are parameters, as in [Al-hassy *et al.*, 2019].

In our implementation we refer to morphisms as views. The type `GView` describes morphisms as defined in 2.3. It consists of a tsource and target theory, as well as the mapping between them. We discussed the combinators we use to build the library in

Section 7.2. Since all the morphisms resulting from these combinators are embeddings, the mapping between theories can be described as a name-to-name map.

```
data GView  = GView {
  source  :: GTheory,
  target  :: GTheory,
  rename  :: Rename }
```

Here `type Rename = Map.Map Name_ Name_` is the type of mapping functions.

8.3.2 Theory Graph Structure

A theory graph consisting of nodes and morphisms is described in Section 2.4. The datatype `TGraph` defines a theory graph as a set of named theories for nodes and a set of named views for edges.

```
data TGraph = TGraph {
  _nodes :: Map.Map Name_ GTheory,
  _edges :: Map.Map Name_ GView }
```

An alternative way to represent graphs would have been to include only the `_edges`, as they contain information about theories. We preferred to keep both mappings to make it easier to lookup theories in the graph.

We noticed that in many cases, the same renames are being reused. So, we also added a `Mapping` type that allows the user to define something like

```
Map plus-zero = {op to + ; e to 0}
```

and reuse it. Accordingly, a library consists of a theory graph and some mappings.

```
data Library = Library {
  _graph   :: TGraph,
  _renames :: Map.Map Name_ Rename }
```

8.3.3 Combinators

Now we describe the implementation of the expressions introduced in Section 8.2 and how they build instance of the type `Library`. The language extension that we introduce to Tog is described in the type `Language`

```
data Language =
    MappingC Name [RenPair]
  | TheoryC Name [Constr]
  | ModExprC Name ModExpr
```

where `MappingC` creates a mapping function, `TheoryC` creates a theory from a list of declarations, and `ModExprC` is the constructor for creating theory expressions. We discuss them in the following sections.

8.3.3.1 Mappings

A definition of a mapping is elaborated into an entry in the `renames` list of the library.

```
addMapping :: Name -> [RenPair] -> Library -> Library
addMapping nm rens =
   over mappings (Map.insert (nm^.name) (renPairsToMapping rens))
```

`over` is the setter function we get by using Haskell lenses. It sets the `mappings` field

of the library to a new instance of `Map` that adds the new mapping to the ones in the input library.

8.3.3.2 Flat Theories

Given a theory presentation as a list of declarations, we construct the new theory and add it to the list of theories in the graph without any morphisms connecting them to other theories.

```
theory :: Name -> [Abs.Constr] -> Library -> Library
theory nm cList =
  let newThry  = GTheory cList waistNm
  in  over graph (over nodes (Map.insert (nm^.name) newThry))
```

8.3.3.3 Theory Expressions

The syntax for the theory expression is introduced in Section 8.1. We now discuss their implementation. We start with the function `updateGraph` which adds theories and morphisms to the graph:

```
updateGraph ::   Name_ -> Either GView PushOut -> TGraph -> TGraph
updateGraph nm (Left view) =
  over nodes (Map.insert nm (target view)) .
  over edges (Map.insert ("To"++nm) view)
updateGraph nm (Right ut) =
  over nodes (Map.insert nm (target $ uLeft ut)) .
  over edges (\e -> foldr (uncurry Map.insert) e
                          [("To"++nm++"1",uLeft ut),
                           ("To"++nm++"2",uRight ut),
                           ("To"++nm++"D",diagonal ut)])
```

The first argument to `updateGraph` is the name of the new theory. Then, the function expects the morphisms resulting from the combinator to be added to the graph. We know that all the combinators compute only one new theory. But, the number of computed morphisms is different based on the combinators. `extends` and `rename` generates one morphism, while `combine` generates three. We capture this with the type `Either GView PushOut`, where `Pushout` is defined as

```
data PushOut = PushOut { -- of a span
  uLeft    :: GView,
  uRight   :: GView,
  diagonal :: GView,
  apex     :: GTheory } -- common point
```

The names of the new morphisms are generated based on the names of the new theories. Since a new theory with a user-given name is defined every time, we know

that the new morphism names have not been generated before.

The functions `computeExtend`, `computeRename`, and `computeCombine` calculate the new morphisms and theories.

1. Computing Extension The inputs to the extension operation is the theory being extended and the new declarations. The new theory is obtained by concatenating the new declarations to the ones already in the theory, given that there is no name clashes between new constructs, and that they are well-typed in the context presented by the theory declarations.

The resulting view has the input theory as source and the computed theory as target. The identity mapping is computed using the `validateRen` function, which assigns a mapping to every symbol in the input theory. In the case of extension the mapping is the identity.

```
computeExtend :: [Constr] -> GTheory -> GView
computeExtend newDecls srcThry =
  GView srcThry (extThry newDecls srcThry) (validateRen srcThry Map.empty)

extThry :: [Constr] -> GTheory -> GTheory
extThry newConstrs thry@(GTheory constrs wst) =
  if List.intersect newConstrNames (symbols thry) == []
  then GTheory (constrs ++ newConstrs) wst
  else error $ "Name clash detected!"
    where newConstrNames = map getConstrName newConstrs
```

2. Computing Rename Computing renames requires computing substitutions. This requires traversing the internal representation of the theory and performing substitution as needed. We use Haskell's scrap-your-boilerplate package [Scrap Your Boilerplate, 2019], based on [Lämmel and Jones, 2003], to perform the traversal. The substitution is then performed using the `gmap` function.

```
gmap :: (Generics.Typeable a, Generics.Data b) => (a -> a) -> b -> b
gmap r x = Generics.everywhere (Generics.mkT r) x
```

`gmap` traverses an instance of type `b` changing every instance of `a` according to the input function `r`. `computeRename` uses `gmap` to perform substitution to declarations of the input theory, as follows

```
computeRename :: Rename -> GTheory -> GView
computeRename namesMap thry =
  GView thry (renameThy thry namesMap) (validateRen thry namesMap)


renameThy :: GTheory -> Rename -> GTheory
renameThy (GTheory constrs wst) m =
  GTheory (gmap (mapAsFunc m) constrs) wst
```

3. Computing Combine The algorithm to compute the result of combining two embeddings work as follows:

- Given the name of the source theory and the two theories to be combined, the first step is to lookup the paths from the source to the target theory. The type `Path` is defined as a non-empty list of `GView`. The function `getPath` searches

the graph for a path between given source and target theories. It starts at the target node and goes backwards, exploring the possible paths until it finds the source. Because none of the combinators result in backward morphisms, we know the theory graph has no cycles. Therefore, this simple search for a path algorithm works. The two paths are used to construct two instances of `QPath`.

```
data QPath = QPath {
  path :: Path,
  ren  :: Rename }
```

- At this point we have the two embeddings and the two rename functions. The next step is to check the preconditions of `combine` as in equation 7.2.1. The function `checkGuards` checks that all symbols in the source theory are mapped to the same symbol after applying the rename function. The scope checker of tog ensures the backwards direction of the equivalence in equation 7.2.1. If the two instances of `QPath` passes the precondition, the pushout can be computed.

- The result theory is computed by taking the disjoint union of the declarations in the source theory, the one on the left of the diamond (the first argument), then the one on the right. Note that this operation is not commutative. If we take the disjoint union of the source, right, then left theories, we get an equivalent but not equal theory. The order of declarations will be different, but the two theories will have the same declarations.

```
newThry =
  GTheory (disjointUnion3 (declarations srcMapped)
                          (declarations lThry)
                          (declarations rThry))
          (waist srcMapped)
```

- The source and target of the resulting morphisms are easy to figure out. The function `allMaps` calculate the mappings by composing the mappings in the views on the path between the two theories, and then the one described by the rename function.

```
lView = GView lt newThry $ validateRen lt (allMaps left)
rView = GView rt newThry $ validateRen rt (allMaps right)
diag  =
  GView commonSrc newThry $ validateRen commonSrc (allMaps left)
```

Chapter 9

The Generator

Graph theories to generated constructions

The flattener compiles theory expressions into a graph of flat theories and morphisms between them. The generator uses the flat theories and manipulate them in order to generate some constructions that are useful when working with algebraic structures. The algorithms used to generate these constructions correspond to how universal algebra defines the constructions.

In Section 9.1, we discuss the requirements of a generation framework that is capable of generating these constructions. In Section 9.2, we present how we have handled those requirements using Tog as the object language. We discuss the generation of the constructions in Section 9.3. We further discuss our approach in Section 9.4.

9.1 Generation Framework

Generating information based on the content of a theory requires dealing with theories as data, and therefore working at the meta-level. Meta programming frameworks differ in their capabilities. We lay out here some basic operations that are required to manipulate theories based on universal algebra definitions. Section 9.1.1 discusses how theories are preprocessed so information can be generated from them. The requirements needed to generate this information are presented in details in Section 9.1.2 and summarized in Section 9.1.3.

9.1.1 Equational Theories

The graph theories are instances of `GTheory` type; see Section 8.3, which is a representation of telescopes as in Equation 2.1.1. In this representation, declarations of a theory are represented as members of the `[Constr]` type, where each `Constr` has a name and an expression denoting its type. Universal algbera has a different representation of theories, which is discussed in Section 3.1. It separates declarations that describe sorts, functions symbols, and axioms. The first requirement to be able to process theories based on universal algebra definitions is to be able to classify theory declarations into these three groups.

9.1.2 Constructions

After presenting a theory as described by universal algebra, it can be used to generate the constructions we present in Section 3.2 and possibly more. We implemented

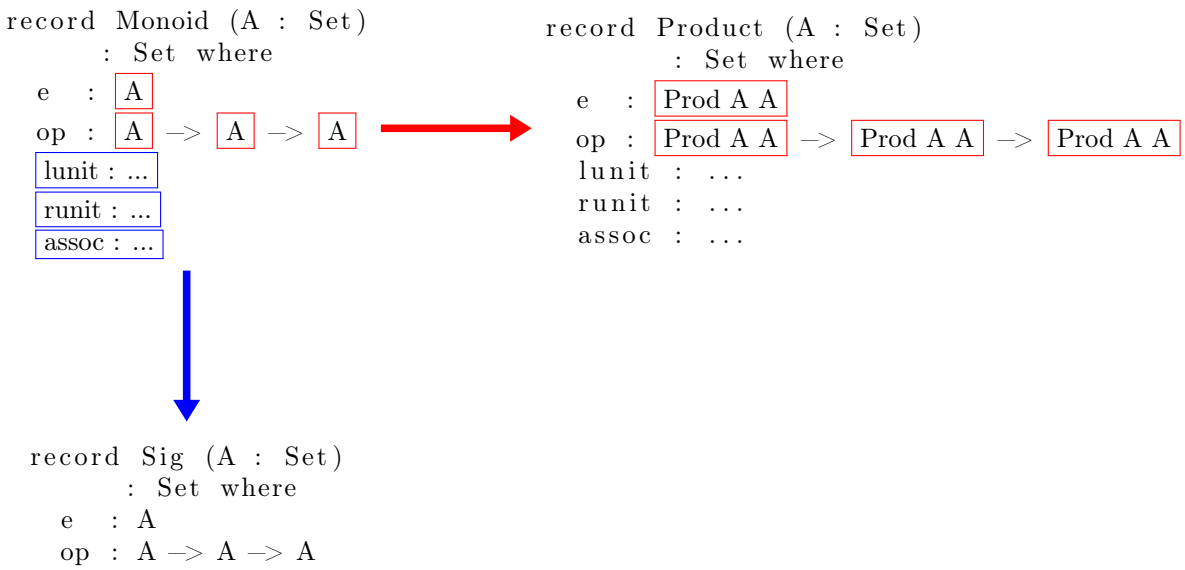


Figure 9.1: Manipulating `Monoid` theory presentation to generate its signature and product theories.

the generation of signatures, product theories, homomorphisms, relational interpretations, and term languages as well as some functions to operate on them. We discuss the requirements of generating each of them in the sequel of this section, where constructions with similar requirements are discussed together.

9.1.2.1 Signatures and Product Theories

Figure 9.1 shows how the theory presentation of `Monoid` can be manipulated to generate signature and product theories. When computing the signature of a theory, one only needs to drop the axioms. Product theories are obtained by replacing every occurrence of the sort `A` with the sort `Prod A A`. To generate them we need to be able to alter the definitions in a theory, by dropping and by substitution.

9.1.2.2 Homomorphisms and Relational Interpretations

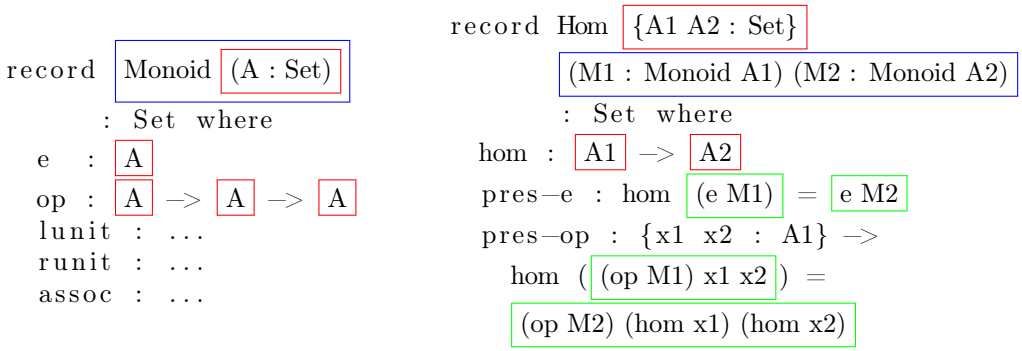


Figure 9.2: Manipulating the `Monoid` theory presentation to generate its homomorphism.

Homomorphisms are structure-preserving mappings between the carriers of two algebras. Relational interpretations are structure-preserving relations between them. Figure 9.2 shows how components of the definitions of `Monoid` are used to generate its corresponding homomorphism theory. To generate them we need the following:

- A representation of 2 instances of a theory with the necessary bindings to define these instances.
- A function/relation between elements of the carriers of the two instances.
- Preservation axioms for every function symbol. To generate these axioms, we need the following:
  - Projection fields of the instances. The names of these projections are qualified if they are fields within a record. Otherwise, they are unqualified.
  - Given a function symbol of the theory, with information about its name and type, a representation of function application of that symbol.

```
-- Basic
data MonoidTerm  : Set where
  op  : MonoidTerm →
        MonoidTerm →
        MonoidTerm
  e   : MonoidTerm
```

```
-- Basic Open
data OpMonoidTerm (n  : Nat)
          : Set where
  v  : Fin n → OpMonoidTerm n
  op : OpMonoidTerm n →
       OpMonoidTerm n →
       OpMonoidTerm n
  e  : OpMonoidTerm n
```

```
-- Closed
data ClMonoidTerm (A  : Set)
          : Set where
  sing : A → ClMonoidTerm A
  op   : ClMonoidTerm A →
         ClMonoidTerm A →
         ClMonoidTerm A
  e    : ClMonoidTerm A
```

```
-- Open
data OpMonoidTerm2 (n  : Nat) (A  : Set)
         : Set where
  v : Fin n → OpMonoidTerm2 n A
  sing : A → OpMonoidTerm2 n A
  op   : OpMonoidTerm2 n A →
         OpMonoidTerm2 n A →
         OpMonoidTerm2 n A
  e    : OpMonoidTerm2 n A
```

Figure 9.3: The term language of `Monoid` expressed in 4 different ways.

9.1.2.3 Term Languages

Some of the constructions we generate for a theory are term languages. We differentiate between 4 different forms of term languages that differ in their expressive power, as shown in Figure 9.3.

The `Basic` term language defines expressions created using the function symbols of the theory. At this level of abstraction, referring to elements of the carrier is not possible. Considering, for example, the binary operation of the basic `Monoid` term language in Figure 9.3, `op`. Its arguments are either the constant `e` or another call for `op`. `Closed` term languages solve this problem by providing the `sing` constructor, abbreviation for *singleton*, that lifts an element of type `A` to an instance of the closed term language. Assuming that the carrier is the type of natural numbers `Nat`, a

possible term in the language would be `op (sing (suc zero)) e`.

`Open` term languages provides extra constructor to represent variables, represented using the `Fin` type. The two open term languages are shown on the right hand side of Figure 9.3.

For all 4 versions, we need to generate a constructor for every function symbol of the theory. The term languages that support referring to elements of the carrier would have an extra constructor for singleton elements with arguments of type `A`; and those that support language with variables would have an extra constructor whose arguments has the type `Fin n`.

The requirements for generating these term languages is:

- A representation for inductive types and constructors.
- A representation of types for natural numbers `Nat`, finite sets `Fin`, and vectors `Vec`.

9.1.2.4 Functions on Term Languages

After generating the term languages, we generate functions that manipulate them. These functions are simplifier, evaluator, induction principle, staged term languages, and staging using representation type that abstracts over the stage. Generating these functions requires the following:

- Pattern matching on the constructors of the term language.
- Constructing recursive calls on the arguments of the pattern.

9.1.3 Requirements

We summarize the different points presented in this section as requirements for the generation framework:

1. A representation of equational theories.
2. A mechanism to manipulate the declarations in a theory, by dropping and by substitution.
3. A representation of instances of a theory.
4. A mechanism to project fields of the instances.
5. A function/relation between elements of the carriers of the two instances.
6. A representation of function application of this field.
7. Computing patterns and recursive calls of a function declaration.
8. Creating inductive types and their constructors.
9. A representation of natural numbers, finite sets and vector types.

9.2 Tog Infrastructure

The generative framework we present here uses Haskell as a meta language to generate construction in Tog, the object language. We mainly manipulate the types in Figure 6.1. In this Section we discuss how we implement the requirements that we laid out in the previous section.

We create a type `EqTheory` to describe first order equational theories as in Chapter 3. In Secion 9.2.1 we describe this type, our definition of an instance of a theory, and how we project fields of an instance. This Section covers requirements 1, 3, and 4. Note that requirement 2 for substitutions is done using the `gmap` function built for the `rename` combinator as explained in Section 8.3.3.

In Section 9.2.2 we describe how we implement requirements 5, 6 and 7, which are related to function definitions and applications. Requirements 8 and 9 are discussed in Sections 9.2.3 and 9.2.4.

9.2.1 Equational Theories

An equational theory in universal algebra abstracts over theory presentations of algebraic structures and consists of a sort, a list of function symbols and a list of axioms, as discussed in Section 2.2. We capture this definition of equational theories by the type `EqTheory`.

```
data EqTheory = EqTheory {
  _thyName    :: Name_  ,   -- the name of the theory
  _sort       :: Constr ,   -- the sort of the theory
  _funcTypes  :: [Constr],  -- the set of function symbols
  _axioms     :: [Constr],  -- the set of axioms
  _waist      :: Int }      -- the number of parameters
```

The `waist` is used in the same way as in `GTheory` from Section 8.2.

Instances of Theories

We define a representation of an instance in terms of its name, the bindings that constitute its parameters and the expression representing the type of this instance.

```
type EqInstance = (Name_,[Binding],Expr)
```

For example, an instance `m : Monoid A` would be represented as:

```
(m,[A : Set], Monoid A)
```

Instances are computed by the function `eqInstance`, where the second argument is used to index the instance in cases where more than one is needed. In this case, both the name of the instances and the names of the bindings are indexed using the input number.

```
eqInstance :: EqTheory -> Maybe Int -> EqInstance
eqInstance thry indx =
  let iname i = twoCharName (thry ^. thyName) i
      binds i =
        let bs = map fldsToHiddenBinds (args thry)
        in if i == 0 then bs else indexBindings i bs
      expr i =
        let bnames = getBindingsNames (binds i)
        in App $ mkArg (thry ^. thyName) : map mkArg bnames
  in case indx of
   Nothing -> (iname 0, binds 0, expr 0)
   Just i  -> (iname i, binds i, expr i)
```

The value of `expr` denotes the type of the instance by applying the name of the theory

to the bindings.

Projecting Fields

Based on whether a declaration of a theory components is a parameter or a field, referring to it will be different. For an instance `m : Monoid A` of a `Monoid` theory that has the carrier as the only parameter, one would refer to the carrier with its name `A`, but would refer to the constant `e` of theory as `m.e`. We provide the function `projectConstr` to compute the projection of one of the declarations of a theory.

```
projectConstr :: EqTheory -> EqInstance -> Constr -> Expr
projectConstr thry (instName,binds,_) c@(Constr n _)  =
  if isArg thry c then App [mkArg $ findInBindings binds c]
  else App [mkArg (n ^. name),mkArg instName]
```

Checking whether the declaration is a an argument or a field is done by the function `isArg` that checks for the `waist` of the `EqTheory`.

A variant of `projConstr` is the function `applyProjConstr` projects the declaration and applies it to variables based on its arity. Its return type is `([Binding],Expr)` where `[Binding]` represents the variables to which the declaration is applied

9.2.2 Functions

A function symbol has the type `Constr` which consists of its name and an expression describing its type. We use the type `FApp` to describe the application of this function symbol to some variables.

```
type FType = Constr
type FApp = ([Binding],Expr)
```

The `[Binding]` in `FApp` refers to the variables the function is being applied to.

A function application is generated by `fapp`. The types of all the bindings is set to be the sort of the theory. The expression is the name of the constr applied to its arguments

```
fapp :: FType -> FApp
fapp (Constr n typ) =
 let nm = n ^. name
     arity = farity typ
     vars = genVars arity
 in if (arity == 0) then ([],App [mkArg nm])
    else ([HBind (map mkArg vars) (etyp typ)],
          App $ mkArg nm : map mkArg vars)
```

The arguments of the functions are generated using the `genVars` function and are used to create the bindings and the function application expression.

When generating functions that manipulate the terms of the theory, like simplifiers and evaluators, one need to pattern match on the function symbols. One common operation on functions is pattern matching. We define the type class `MkPattern` and its two instances for `FType` and `Expr`.

```
class MkPattern a where
  mkPattern :: a -> Pattern
```

The pattern depends on the arity of the function symbol, and is generally the

application of the name of the function symbol to its parameters.

```
if (arity == 0)
then IdP $ mkQName nm
else ConP (mkQName nm) $ map (IdP . mkQName) vars
```

The functions `functor` and `functor'` support applying a functor to an expression, be it the name of a function symbol or a more complex expressions.

9.2.3 Datatypes

Some of the constructions that can be generated from theory presentations are represented as datatypes, like term languages. A datatype in Tog has the type `Decl`. The type `DTInst` captures the instances of a datatype in the same way as `EqInstance`. Similar to functions, we deal with datatypes in two different forms, definitions and instances.

```
type DTDef = Decl
type DTInstance = (Name_,[Binding],Expr)
```

Instances are computed by the function `tinstance`. The bindings are computed based on the parameters of the datatype. The expression denoting the type of the instance is computed by applying the name of the datatype to the bindings used the `App` constructor.

9.2.4 Prelude Definitions

The constructions that we defined here depends on some definitions that act as the prelude of the library. We define these as literals:

```
nat :: [String]
nat =
  ("data Nat : Set where { " ++
     "zero : Nat ;" ++
     "suc  : Nat -> Nat }") : []
```

These strings are parsed by the function `parseDecl`, which turns it to a Tog definition of type `Decl`.

9.3 Constructions For Free!

By providing the appropriate tools to operate over the internal syntax of Tog, we are ready to generate the universal algebra constructions related to equational theory presentations. In the following sections we describe the generation of these constructions.

9.3.1 Signature

Signatures represent the language of the theory, without any properties governing them. It is common in mathematics to talk about algebras over some signature. Signatures are obtained from theory presentations by dropping axioms. In Tog, the process of generating the signature is done in 3 steps via the `signature_` function.

```
signature_ :: Eq.EqTheory -> Eq.EqTheory
signature_ = set Eq.thyName ("Sig") . set Eq.axioms [] .  gmap ren
```

The function `ren :: Name -> Name` renames the fields of a theory by adding a suffix

`"S"`. This is needed because the Tog scope checker would not accept overloaded names of fields within the same module. In case the code is exported into a system that supports this kind of overloading, the suffix can be removed. Note that this rename will be needed when generating any new construction. `gmap` function traverses the `EqTheory` applying `ren` whenever a `Name` type is encountered. The `Eq.axioms` list is set to be empty, dropping the axioms of the theory.

9.3.2 Product Algebra

Product algebras group together algebras of the same theories. The type `Prod` lifts a type `A` to a type `Prod A A`, where `Prod` is standard product type. The lifting of the type `A` is done via substitution of every `A` with `Prod A A`. The function `productThry` uses this type to compute the product theory

```
productThry :: Eq.EqTheory -> Eq.EqTheory
productThry t =
 let sortName = getConstrName (t ^. Eq.sort)
 in set Eq.thyName ("Product") $
    gmap (prodType sortName) $
    gmap (ren sortName) t
```

The `prodType` function does the type lifting for the sort as follows:

```
prodType :: Name_ -> Expr -> Expr
prodType sortName (App [a]) =
  if (getArgName a) == sortName
  then App [mkArg "Prod", a, a] else App [a]
prodType _ x = x
```

9.3.3 Homomorphism

Theories are presented as record declarations in Tog, and so are their homomorphisms. The following function generates the homomorphism declaration:

```
homomorphism :: Eq.EqTheory -> Decl
homomorphism thry =
  let nm = "Hom"
      i1@(n1,b1,e1) = Eq.eqInstance thry (Just 1)
      i2@(n2,b2,e2) = Eq.eqInstance thry (Just 2)
      fnc = homFunc thry i1 i2 (thry ^. Eq.sort)
      axioms = map (presAxiom thry i1 i2 fnc) (thry ^. Eq.funcTypes)
  in Record (mkName nm)
   (mkParams $ b1 ++ b2 ++
               map (\(n,e) -> Bind [mkArg n] e) [(n1,e1),(n2,e2)])
   (RecordDeclDef setType (mkName $ nm ++ "C")
                  (mkField $ fnc : axioms))
```

i1 and i2 are the two instances of `thry` created using `eqInstance` as described in the

Section 9.2.1. Those instances are used to create the parameters of the homomorphisms using the `mkParams` function.

The declarations of the homomorphism record are the homomorphism function and the preservation axioms. The function is generated by `homFunc` which uses the function `projectConstr`, described in Section 9.2.1, to project the carriers of the two instances.

```
homFunc :: Eq.EqTheory -> Eq.EqInstance -> Eq.EqInstance -> Constr
homFunc thry i1 i2 =
  let carrier = thry ^. Eq.sort
  in Constr (mkName homFuncName) $
       Fun (Eq.projectConstr thry i1 carrier)
           (Eq.projectConstr thry i2 carrier)
```

Equations of the preservation axioms are generated by the `equation` function. It uses `applyProjConstr`, explained in 9.2.1, which give the expression of function application as well as list of the variables its applied to.

```
(bind1,expr1) = Eq.applyProjConstr thry i1 constr Nothing
    (_,expr2) = Eq.applyProjConstr thry i2 constr Nothing
```

These pieces are used to construct the `Pi`-type as follows

```
Pi (Tel bind1) $ Eq (lhs homFunc expr1) (rhs homFunc expr2)
```

9.3.4 Relational Interpretation

A relational interpretation is a structure preserving relation. We discuss it in Section 2.6. Its implementation looks very similar to that of homomorphism — a structure preserving function. Some of the similarities are that they are both records and have the same parameters. However the fields are different. Instead of having a function between the two carriers, we have a relation. The function `mkInterpType` generates the type of the relation field. It looks very similar to the function `homFunc`, except the type is a relation from carriers to the type `Set` as follows:

```
Fun (Eq.projectConstr thry i1 carrier) $
    Fun (Eq.projectConstr thry i2 carrier) setTypeAsId
```

Then, we generate the axioms that guarantees preserving structure. For a binary operation, this axiom would look as follows

```
interp-op : {x1 x2 : A1} {y1 y2 : A2} →
          interp x1 y1 → interp x2 y2 →
          interp (op x1 x2) (op y1 y2)
```

To generate these axioms, we call `applyProjConstr` to get the bindings and the function application expression, the same as done in homomorphism generation. Then, we align them into lists of the form `[x1,x2,op x1 x2]` and `[y1,y2,op y1 y2]`. The elements of the lists are used to create the axioms by applying the relation on the corresponding elements from the two lists

```
zipWith (\x y -> App [mkArg (interpName^.name),x,y]) args1 args2
```

9.3.5 Term Algebras

We capture the 4 forms by the type `Term`.

```
data Term = Basic
          | Closed CarrierName
          | BasicOpen NumOfVars
          | Open NumOfVars CarrierName
```

The arguments to the constructors reflect the arguments of the term language in every case. For example, the type `OpMonoidTerm2` in Figure 9.3 has the type `Open "n" "A"`.

In Section 9.3.5.1 we discuss the generation of the 4 different forms of term languages. We also generate some functions related to the term languages; functions for simplifying terms of the language, evaluating them, constructing the induction principles, and constructing the staged version of the term language. To generate these functions, we need to generate their types and the definitions which consists of patterns and expressions evaluating the value of the functions at the given pattern. The types of the functions are generated by implementing a `typeSig` function for each of them. Each of their declarations has the form:

```
FunDef Name [Pattern] FunDefBody
```

The patterns and expressions of every declaration is defined using the `patternsExprs` function. In cases when there is more than one argument, some adjustments to the patterns and/or expressions may be needed which are defined within the `adjustPatterns` or `adjustFunCalls` functions. Finally, each one of the 4 forms of a term language will have its own `oneX` function that generates the function `X`. These functions serve as the interface for defining functions on the term language. We describe each one of

these functions in Sections 9.3.5.2 – 9.3.5.5. We also generate a staged version of the term language based on a representation type, which we discuss in Section 9.3.5.6.

9.3.5.1 Term Language

A term language represents the type of terms described by the theory. We use the `TermLang` type to represent term languages.

```
data TermLang = TermLang {
  termTy  :: Term,     -- One of the 4 forms
  tname   :: Name_,    -- The name of the term language
  params  :: Params,   -- The parameters of the type
  cons    :: [Constr] -- The constructors of the type
}
```

Starting from an `EqTheory`, the function `tlang` generates a `TermLang`. The parameters are decided depending on the value of `termTy`. The constructors of the type are declared based on both the type of the term and the fields of the theory. A `Closed` or `Open` term language would have constructors for constants

```
Constr (mkName singConstrNm)
    (Fun (App [mkArg carrierNm]) declType)
```

A `BasicOpen` and `Open` term languages would have constructors for variables.

```
let fin = App [mkArg "Fin", mkArg natVarNm]
in Constr (mkName vconstrNm) (Fun fin expr)
```

In all cases, a constructor is generated for every function symbol of the theory

```
constrs = map (constructorsHelper $ termType thryNm t) cs
```

where `cs` is the list of fields of the theory, `termTyp` generates the type of one argument of the function by calling `liftType'` that applies the name of the type to its arguments. `constructorsHelper` repeats this type for as many times as needed for the type of the constructor.

9.3.5.2 Simplifiers

Some simplification rules can be generated from theory presentations based on the axioms, i.e. rewriting some terms into simpler forms based on equality axioms. For every term language of a theory, we generate a simplification function based on this idea.

For a term language `L`, the simplifier has the type $\texttt{L} \rightarrow \texttt{L}$. In cases when `L` is parametrized, the type is preceded by the bindings. The bindings and the type expressions are computed by calling `tlangInstance` which uses the `tinstance` function described in Section 9.2.3. The construction of the type aftewards is straightforward as follows

```
(_,binds,typApp) = tlangInstance tl
typeExpr Basic = Fun typApp typApp
typeExpr _     = Pi (Tel binds) (Fun typApp typApp)
```

The simplification rules are then generated by the `simpRules` function. For every equation $\texttt{t}_1\texttt{= t}_2$, the simplifier need to decide if any of the two terms of the equations is simpler than the other. For this purpose, a well-founded ordering relation is needed. We choose a very simple relation that produces a basic simplifer, i.e. we do not

guarantee to reach the simplest form of the term. The relation we use is the length of the term in the sense of its number of literals, computed using the following expression:

```
explength :: Expr -> Int
explength e = everything (+) (mkQ 0 $ \ (Name _) -> 1) e
```

The function `minMax` make sure the expressions are oriented in the right way:

```
minMax :: Expr -> Expr -> Maybe (Expr,Expr)
minMax e1 e2 =
  if (explength e1 == explength e2) then Nothing
  else if explength e1 < explength e2 then Just (e1,e2)
  else Just (e2,e1)
```

The longer term is converted to an element of type `Pattern` and used as input to the simplifier that maps to the shorter term.

```
simpRules :: EqTheory -> Term -> [(Pattern,Expr)]
simpRules thry term =
 let mpng = Map.toList $ mapping thry term
     axms = map (foldrenConstrs mpng) (thry ^. axioms)
 in mapMaybe simplify axms
```

For simplification to be effective, one needs to traverse the expression looking for subexpressions that can be simplified. The declarations that does the traversals is generated by `simpDecls`. For each constructor, the function generate a pattern using `mkPattern` and a term using the `fapp` function as discussed in Section 9.2.2. The recursive calls on the arguments of the expression is generated by calling `adjustFuncCalls`.

```
simpDecls :: Term -> [Constr] -> [(Pattern,Expr)]
simpDecls term ftyps =
   zipWith ((,)) patterns fundefs
   where patterns = map mkPattern ftyps
         fundefs = map (functor' (adjustFuncCalls term) . fappExpr) ftyps
```

Lastly, if there are singleton or variable constructors they need to be returned as is using the `simpVarsConsts` function.

```
simpVarsConsts :: [Constr] -> [(Pattern,Expr)]
simpVarsConsts cs =
  zipWith ((,)) (map mkPattern cs) (map fappExpr cs)
```

The declarations of the simplifier is the result of concatenating all these declarations as follows

```
simpRules thry term
++ simpDecls term (filter (not . isConstOrVar) cs)
++ simpVarsConsts (filter isConstOrVar cs)
```

Note that we do not generate the simplification functions for the `Basic` term language. In some cases, like in `Magma`, the `Basic` term language does not have a base case, and therefore a termination proof of the simplification function is not trivial. Some theorem provers, like Lean, would not accept this definition.

9.3.5.3 Evaluators

The evaluator generates 4 functions, one for every term language. In the simplest case, the `Basic` term language, the evaluation function for `Monoid` will have the following type

```
evalB : {A : Set} → Monoid A → MonoidTerm → A
```

An expression of type `MonoidTerm` is evaluated to an element of a carrier `A` on which a monoid structure exists. The constructors of the language is mapped to operations of the theory, in a way opposite to what was done to generate the term language. Therefore, the function that generates the evaluator needs to deal with both the equational theory and the term language. The types of the evaluator functions are generated by the `ftype` function. The first step is to generate the definition of instances of both

```
(eqbind,eqinst) = eqInstance thry Nothing
(tbind,tinst) = tinstance (tlToDecl termlang) Nothing
newBinds = unionBindings eqbind tbind
```

The functions `eqInstance` and `tinstance` generate instances of the theory and the term languages, as explained in Sections 9.2.1 and 9.2.3. Both instances might be parameterized, in which case some bindings need to be defined before they can be declared. Those bindings are defined in `eqbind` and `tbind`. The bindings of the function are the union of the two bindings.

Function declarations are defined for variables, constants and function symbols. In case of variables, a call to the `lookup'` function is performed as follows

```
[FunDef (mkName $ evalFuncName term) -- call for vars
        (concatMap (cpattern instName term) vs)
        (lookup' envName)
        | not (null vs)]
```

where `vs` is the list of variable declarations. The function `cpattern` creates the pattern for a constructor. `lookup'` creates a call to the `lookup` function. Creating the function declaration for constants look very similar, except it returns the constant itself.

```
[FunDef (mkName $ evalFuncName term) -- call for constants
         (concatMap (cpattern instName term) constants)
         constFunc
         | not (null constants)]
```

For every other constructor in the type, a pattern of it is created using `cpattern` and assigned to one of the declarations of the theory using `funcDef`.

```
zipWith (FunDef (mkName $  evalFuncName term))
        (map (cpattern instName term) tDecls)
        (map (funcDef eq instName term) eqDecls)
```

The value of the expression at each constructor is mapped to the corresponding function symbol in the same order. This make sense as we deal with theories as telescopes, and so order matters. When we generate the term language we do not change the order. Now that we are assigning back those declarations, the order is used to map them back.

9.3.5.4 Induction Principle

Induction principles are defined over sets with well-founded relations. In the case of structural induction, they are based on the subterm relation. Structural induction requires a base case, a constant or variable symbol in the language. In cases like the `Basic` term language of magma, the variables `x` and `y` of an expression `op x y` can only be substituted by other `op` expressions and therefore never gets smaller. One can argue that in this case the induction principle is not defined. Despite that, we run the following experiment in Coq, which automatically generates the induction principle for types declared as `Inductive`.

```
Inductive magma : Set := op : magma -> magma -> magma.
Check magma_ind.
```

The following induction principle is generated

```
magma_ind : forall P : magma -> Prop,
 (forall m : magma, P m ->
    forall m0 : magma, P m0 -> P (op m m0)) ->
 forall m : magma, P m
```

A possible explanation is that the type `magma` is empty, and therefore it's fine to have its induction principle generated. Based on this observation, we decided to generate the induction principle for all term languages.

For every constructor of the term language, we use `fapp` to generate the term resulting from applying this function symbol to some bindings, along with these bindings. To generate the induction principle for a predicate `P`, we need to generate

a type stating that given proofs of `P` applied to the bindings, we can induce the proof of `P` applied to the term. For every constructor the function `typeFun` does that.

```
if null binds then applyPred fexpr
else Pi (Tel binds) $
        curryExpr $ concatMap applyPredToBindings binds
                    ++ [applyPred fexpr]
```

In cases when the term language has a singleton or variable constructor, those ones also need to be included in the type, but their construction is straightforward.

9.3.5.5 Staged Term Languages

Systems that support multi-stage programming (MSP) enables staging the evaluation of expressions between a current (`Now`) stage and a future (`Later`) one. An expression that is staged for a `Later` stage, is dealt with as `Code`. The details of MSP is discussed in Section 2.7.

We generate functions to automatically add staging annotations to terms of the term language of the theory, as follows

- constants (whether elements of the carrier or 0-ary functions) has values at the current stage.
- variables do not have values until runtime.
- A function symbol can be computed if all its parameters have values at compile time.

The generator depends on functions `stage1`, and `stage2` that provides the lifting of unary and binary expressions based on the status of their arguments. In cases when

the expression are annotated for a `Later` stage, one need to be able to talk about their `Code` as expressions instead of their values. Therefore, we need functions `codeLift1` and `codeLift2` to lift an expression to its `Code` version. To give more clarity, the type of stage and codeLift function for unary operations are

```
codeLift1 : {A B  : Set }  → (A  → B )  →
                        (CodeRep A s1 → CodeRep B s1)
codeLift1 f (Q x )  = Q (f x )

stage1 : {A B  : Set } → (A → B ) →
                        (CodeRep A s1 → CodeRep B s1 ) →
                        Staged A → Staged B
stage1 f g (Now x)  = Now (f x )
stage1 f g (Later (Computation _ x ))  =  Later (Computation Expr (g x))
```

The `codeLift` functions expects functions of specific arities. We have those functions as declarations within a record, instead of function definitions within the module. Tog does not treat them the same way; therefore we had to generate function declarations for each constructor of the term langauge, in order to pass them to the theory.

The `codeLift` and `stage` functions interplay together as follows:

```
case exprArity expr of
  0 -> App [mkArg "Now",mkArg $ n ^. name]
  1 -> stageH "stage1" "codeLift1"
  2 -> stageH "stage2" "codeLift2"
  _ -> error "Cannot stage term, provide a staging function"
```

9.3.5.6 Representation types

Inspied by the tagless language embedding technique [Carette *et al.*, 2009], we use representation types to abstract over stages. Consider the following type based on the term language of `Monoid`

```
record StagedRepr (A : Set) (Repr : Set → Set) : Set where
 constructor repr
 field
   opT : Repr A → Repr A → Repr A
   eT  : Repr A
```

By instantiating `Repr` type with `Staged`, we can get the staged type for the terms of `Monoid` as

```
taglsMon : StagedRepr MonoidTerm Staged
taglsMon = record {eT = Now e ; opT = stage2 op (codeLift2 op)}
```

The type `Repr` is defined internally as:

```
Bind [mkArg reprTypeName] $ Fun (App [mkArg "Set"]) (App [mkArg "Set"])]
```

The fields of the record are all generated by the following expression

```
map (liftConstr reprTypeName) fdecls
```

where `liftConstr` would lift a type `A` into `Repr A`.

9.4 Discussion

Knowledge representation is a key part of a generation framework. Our representation of `EqTheory` follows from the axiomatic representation of algebraic structures as presented in universal algebra. The definitions of `FType` and `DType` corresponds to the representation of functions and datatype, respectively, in Tog. Less obvious was the representations of instances `EqInstance` and `DTInstance` and function application. Once the knowledge capture and utility functions presented in Section 9.2 are in place, generating new constructions becomes a straight forward task.

Another useful lesson we learn here is about the importance of having a strong and small core language for manipulating structures. Many things were easy to do in Tog because it is a small system. But we also faced difficulties due to the immaturity of some features in Tog. For example, the generated definition of induction is not accepted by Tog's type checker if the hidden argument `{p}` is not passed explicitly. Most feature-rich systems, like Agda and Lean, will not need to have this argument defined. Another needed feature is treating constructors as functions, where they can be passed to higher order functions. Tog does not support that, although many systems do.

Using this framework we are able to generate a library of 106468 lines of code from the representation of 227 theories. Appendix B shows the generated definition for `Monoid` theory. All the generated files are present on github under https://

github.com/ysharoda/Deriving-Definitions/tree/115462d85389/Library/generated.

Chapter 10

The Exporter

Generated constructions to proof assistants

Generating the definitions of constructions from a theory presentation saves a lot of library development time, but having these definitions in a feature-rich language makes it even more useful. In this chapter we implement an automatic translator of the library theories and their related constructions to Agda and Lean. This part is related to the third research question from Section 1.1.

We study how different Agda and Lean are from Tog in Section 10.1. We discuss our design of an exporter in Section 10.2. The implementation in Haskell, the meta-language for Tog is disucssed in Section 10.3. We compare our generated Agda code to the one in the Agda standard library [Agda Library, 2020] and discuss how close we can get to the standard library presentation in Section 10.4. We end by a discussion in Section 10.6.

10.1 Beyond Tog

As an experimental small language, Tog lacks some features that are usually found in main stream ones. In this section we discuss these features

10.1.1 Universes

Tog provides only one kind `Set`. It does not support a universe hierarchy, and so in Tog `Set : Set`. On the other hand, Agda and Lean have an infinite number of universes. This is expressed in Agda as $\texttt{Set}_{\texttt{n}}$: $\texttt{Set}_{\texttt{n+1}}$ for any natural number n.[1] All the constructions we generate belong to the same level, except for relational interpretations, which describe a structure-preserving relation between two instances of the theory; see Section 9.3.4. In Tog, relational interpretations are records and the relation is a field of the record represented as

```
interp : A1 -> A2 -> Set
```

for types `A1` and `A2`, which are carriers of the two instances. A record with this field in Tog has a type `Set` and therefore belong to universe level zero. When exported to Agda or Lean, its definition needs to have the type universe level 1.

10.1.2 Prelude Definitions

The constructions we generate from theory presentations depend on the Tog definitions of `Nat`, `Fin`, `Vec`, and `lookup`. Tog does not support indexed types and defines `Fin` as follows:

[1] In Lean, the hierarchy is expressed as `Type n : Type (n+1)` for any natural number `n`.

```
data Fin (n : Nat) : Set where
  fzero : (m : Nat) (p : n == suc m) -> Fin n
  fsuc  : (m : Nat) (p : n == suc m) (i : Fin m) -> Fin n
```

This leads to a rather complicated definition of the `lookup` function.

On the other hand, Agda supports indexed types, has a simpler definition of `Fin` and `lookup`, and has these definitions in its standard library. Similarly, Lean has types and functions for the same purposes, but with different names.

10.1.3 Equality Check in Pattern Matching

One of the things we generate is a simplifier that uses axioms like `e * x ≡ x` to simplify expressions; see Section 9.3.5.2. In a theory that has a binary operation with an inverse and a unit, like `Group`, a possible axiom is

```
op x (inv x) ≡ e
```

which would give rise to a simplification rule. To perform this simplification, one needs to compare the two occurrences of `x` for equality. Non-linear pattern matching is the case when the same variable name can occur more than once in patterns, in which case the value referred to at these occurrences are considered equal. While Tog accepts non-linear patterns, Agda and Lean are restricted to linear pattern matching and would not accept that code. Therefore, to perform the simplification, we need to compare them using decidable equality.

10.1.4 Functions as Constructors

In Section 9.3.5.5 we discussed automatically annotating term languages to produce staged expressions. We discussed a problem related to how Tog represents constructors of a datatype. Tog does not allow passing constructors to higher order functions. Instead, we had to define a function corresponding to these constructors and pass it to the functions that lifts them to their staged versions. When exporting to Agda or Lean, we do not need to keep this trick.

10.2 Exporter Design

The Tog definitions have all the information needed to mathematically present the concepts they are describing. The process of exporting these definitions from Tog to Agda or Lean can be seen as *presenting* them in a way that the target language understands (type checks).

In the previous section, we discussed some of the misalignments between the presentations of concepts in Tog versus Agda or Lean. The preprocessing manipulates the syntax tree to resolve these issues. Afterwards, the exporter traverses the syntax tree and prints the output in the format accepted by the target language. Language specific keywords and options are specified using a configuration type. The design of the exporter is illustrated in Figure 10.1.

10.3 Implementation

We now discuss our implementation of the design in Section 10.2. We start by discussing the preprocessing functions in Section 10.3.1 showing how they solve the

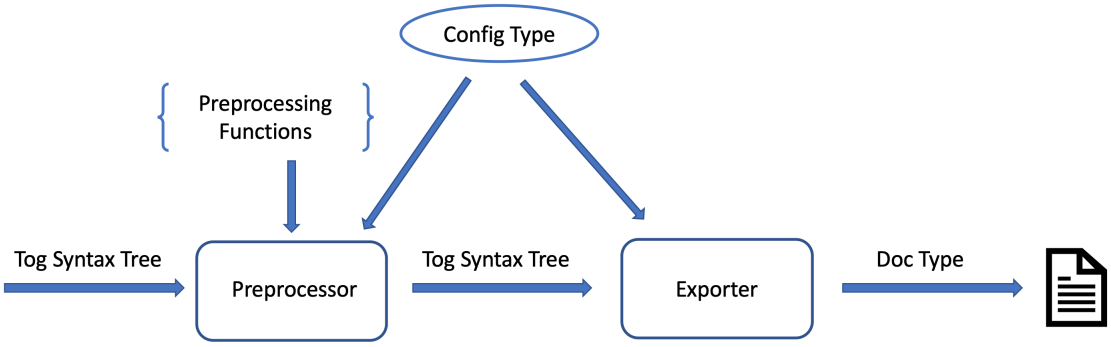


Figure 10.1: The design of the exporter.

problems highlighted in Section 10.1. In Section 10.3.2 we introduce the `Export` type class that performs that translation from the modified Tog syntax tree to the definitions of the target language.

10.3.1 The Preprocessor

The first stage of the exporter is to preprocess the Tog syntax tree to account for the issues discussed in Section 10.1. In this section, we discuss the manipulations performed by the preprocessor.

10.3.1.1 Universes

To solve the universes problem, we provide the function `universeLevel` which checks the fields of a record for a `Set` type. If it finds one, it sets the type of the record to universe level 1, where the representation of the level is read from the config file.

```
universeLevel :: Config -> Fields -> Doc
universeLevel conf flds =
  text $
    if elem "Set" $ everything (++) (mkQ [] (\ (Name (_,x)) → [x])) flds
    then (level1 conf) else (level0 conf)
```

The function `universeLevel` is called every time a record header is printed.

10.3.1.2 The Prelude

Exporting the prelude definitions is done differently than exporting the generated code. The configuration type includes information about how they are processed, via the field `prelude_includes :: Either FilePath ([ImportDecl],[String])`. If the value has the type `FilePath`, then the provided file includes the definitions of the prelude. Otherwise, the configuration provides a list of import declarations, to be added at the beginning of the prelude module, and a list of the names of definitions to be exported. For example, the `prelude_includes` of the Agda configuration is:

```
Right (["open import Agda.Builtin.Equality",
        "open import Agda.Builtin.Nat",
        "open import Data.Fin",
        "open import Data.Vec"]
      ,["Prod","Wrap","Stage","CodeRep","uncode","code","run",
        "Choice","Comp","Staged","expr","const",
        "stage0","stage1","stage2","codeLift1","codeLift2"])
```

The imports for modules other than prelude is defined using the `imports` config

declaration.

The function `mkImport` creates import declarations for all modules based on the information retrieved from the configuration. `import_`, `open_`, and `openimport` are the keywords to import, open, or open-import a module in the target language. The function reads the names of these modules and creates instances of `Decl` using the corresponding constructors.

```
mkImports :: Config -> [String] -> [Decl]
mkImports conf imprts =
  let getNames prefix =
        if prefix == "" then []
        else removePrefix conf $ filter (isPrefixOf prefix) imprts
      createImport x = ImportNoArgs $ mkQName x
  in (map (Import . createImport) $
        (getNames $ import_ conf) \\ (getNames $ openimport conf))
  ++ (map (OpenImport . createImport) $ getNames (openimport conf))
  ++ (map (Open . mkQName) $
        (getNames $ open_ conf) \\ (getNames $ openimport conf))
```

When importing functions, the order of their inputs may be different than that used when calling the same function in Tog. This is the case with the definition of `lookup` in Agda versus Tog where the two arguments are flipped. In Lean the function name is `nth` and the arguments are also flipped with respect to the Tog definition. To solve this problem, every target exporter has a function `callFunc` that adjusts the call to the function. The one for Agda is:

```
callFunc :: Expr -> Expr
callFunc a@(App [nm,_,a2,a3]) =
  if (getArgName nm == "lookup") then App [nm,a3,a2] else a
callFunc e = e
```

`callFunc` is called before every function application is exported. Therefore, it can be easily extended to adjust calls to any function.

10.3.1.3 Simplifier

One of the constructions that can be generated is decidable equality. In case it is generated, it can be used to check for equality of variables. Since we do not generate it, we remove simplification rules that includes two occurrences of the same variable name in the pattern.

10.3.1.4 Functions as Constructors

Not allowing constructors to be passed to higher order functions resulted in creating a function declaration for every constructor of term languages during the generation phase of the interpreter. The function `constructorsAsFunctions` removes these generated functions, as they are not needed for Agda or Lean. The function is defined as follows:

```
constructorsAsFunctionsHelper :: Config -> [Constr] -> [Decl] -> [Decl]
constructorsAsFunctionsHelper conf cs decls =
  let cnames = map getConstrName cs
      toFindNames = map opDeclToFuncName cnames
      mapping = zip toFindNames cnames
  in if(constructors_as_functions conf) then decls
      else foldrenConstrs mapping $
               filter (\d -> not  $  elem (declName d) toFindNames) decls
```

`cnames` is a list of the names of the constructors and `toFindNames` is a list of their corresponding functions. `filter` is used to remove these definitions from the list of declarations of the module. `foldrenConstrs` is then used to substitute their names with the names of the corresponding constructors.

10.3.1.5 Field Names

Another misalignment between Tog, Agda, and Lean is what names can be used for fields. Agda does not allow them to be numbers. Lean does not accept numbers or symbols like `+, *, |>, <|`. We provide a function `replace` that is called before any `Name` is printed. The `replace` function for Agda is:

```
replace :: String -> String
replace nm =
  let pieces = splitOn "_" nm
      cond = \x -> if (x == "0" || x == "1") then x ++ ₍ else x
      postProcess lst = (head lst) : (map ("_"++) $ tail lst)
  in concat $ postProcess $ map cond pieces
```

A name which is just `0` or `1` is concatenated with a suffix ₍. The suffix is also added if the `0` or `1` is part of a name, but is separated by `_`. This accounts for the naming convention of the MathScheme library for axioms.

10.3.2 The Exporter

The type class, `Export`, prints the Tog definitions in a form accepted by the target language, whose type checker is then called on them.

```
class Export a where
  export :: Config -> a -> Doc
```

The `Config` type is used to describe the configuration of each language. It contains details about language specific properties or pieces of syntax. For every type in the Tog AST, we create an instance for the `Export` class. We use the Haskell pretty printer provided by `Text.PrettyPrint.Leijen`, which is an implementation of the pretty printer described in [Wadler, 2003].

10.3.2.1 The Pretty Printer

In [Wadler, 2003], an algebra for defining pretty printers is introduced, based on 6 primitives:

```
(<>) :: Doc -> Doc -> Doc
empty  :: Doc
text :: String -> Doc
line :: Doc
nest :: Int -> Doc -> Doc
layout :: Doc -> String
```

where `Doc` is the type of a document. The `(<>)` operation concatenates two documents. It is an associative operation with `empty`[2] being its right and left unit. On top of these primitives, we have used the following functions provided by the `Text.PrettyPrint.Leijen`.

- `(<+>)` : concatenates two `Doc` instances with a space between them.
- `(<$$>)` : concatenates two `Doc` instances with a `line` in between them.

10.3.2.2 The Exporter Type Class

We present here some of the interesting instances of the `Export` type class[3]. Our generator defines every theory along with its generated constructions in a `Module`. Exporting a `Module` is described by the following instance:

[2] [Wadler, 2003] refers to `empty` as `nil`.

[3] The full code is available at: https://github.com/ysharoda/Deriving-Definitions/blob/7e19c3c7d624/src/Tog/Exporting/export.hs.

```
instance Export Module where
  export conf (Module nm params decls) =
    export conf imprts <$$>
    text (m1 conf) <+> export conf nm <+> text (m2 conf) <+>
    export conf params <+> text (m3 conf (isEmpty decls)) <$$>
    (indent 2 $ export conf defs) <$$>
    moduleEnd conf nm
    where (imprts,defs) = split conf decls
          isEmpty (Decl_ []) = True
          isEmpty _ = False
```

In order to write one exporter with two target languages, we need to investigate the commonalities and differences between them. The first obvious difference is the keywords used. The exporter reads the keywords of the target language from its configuration. On the level of modules, we use the configuration fields `m1, ..., m4` as follows

```
m1 nm m2 params m3
...
m4
```

All configuration fields are printed using the `text` function.

Another difference between our two target languages is the structure of their module system. The general structure of modules in Agda and Lean is

```
module nm params where
  import ...
  ...
```

```
import ...
section nm
  ...
end nm
```

The configuration of each language specifies the position of the import declarations. The function `split` checks the configuration for this information and accordingly splits the module declarations into those to be printed before its header (if any), and those who are part of the module.

The `moduleEnd` function checks if the module needs to be closed with any keywords, and whether the name of the module needs to be included as in Lean. The `export` function is called on the components of the module, which are the name (`nm`), the parameters (`params`) and the declarations within the module (`decls`).

The parameters of a module are represented as `[Binding]`, which can be hidden or explicit. Exporting the binding is done by calling `export` on its arguments and type expression.

```
instance Export Binding where
  export conf binds =
    let arguments as = hsep $ map (export conf) as
        binding x =
          arguments (getBindingArgs x) <+> text (bind_of_type conf)
          <+> export conf (getBindingExpr x)
    in case binds of
      Bind  _ _ -> parens $ binding binds
      HBind _ _ -> braces $ binding binds
```

Every `Binding` consists of a set of arguments of type `Arg` that defines the variables of the binding. The function `getBindingExpr` returns the type of those binding arguments.

The body of the module consists of declarations of type `[Decl]`. Exporting each of these declarations is straight forward by calling the `export` function on its components. Function definitions in Agda are declared by writing the function names for every pattern, while in Lean guards are used to declare the different patterns.

```
f : binds - list → type - expr
f x0 .. xn = ...
...
```

```
def f binds - list : type - expr
| x0 ... xn = ...
...
```

If the function name is part of the definition, the configuration is set to be `"fname"`. The function `funcHeader` takes care of this case.

The instance of `Export` for `Decl` is:

```
instance Export Decl where
  export conf (TypeSig sig) = export conf sig
  export conf (FunDef nm ps body) =
    funcHeader (f5 conf) nm <+> (hsep $ map (export conf) ps)
    <+> text (f6 conf) <+> export conf body <+> text (f7 conf)
    where funcHeader flag fname =
            if flag =="fname" then export conf fname else text flag
  export conf (Data nm ps body) =
    text (d1 conf) <+> export conf nm <+> text (d2 conf)
    <+> export conf ps <+> text (d3 conf) <+> export conf body
    <+> openDatatype conf nm
  export conf (Record nm ps body) =
    text (s1 conf) <+> export conf nm <+> text (s2 conf)
    <+> export conf ps <+> text (s3 conf) <+> export conf body
  export conf (Open imp) = text (open_ conf) <+> export conf imp
  export conf (Import imp) = text (import_ conf) <+> export conf imp
  export conf (OpenImport imp) =
    text (openimport conf) <+> export conf imp
  export conf (Module_ m) =
    linebreak <+> export conf m  <+> linebreak
  export _ _ = empty
```

With instances of `export` for every type in the Tog abstract syntax, a single call to `export` on the top level module of the tog library generates the equivalent Agda

definitions[4].

10.4 Comparison With Agda Standard Library

We compare the the definitions that we generate in Agda to those in its standard library, highlighting how close we can get to them.

Algebraic structures in Agda are unparameterized records. For example, `Monoid` is defined as[5]:

[4]The generated files available at: https://github.com/ysharoda/Deriving-Definitions/tree/master/Library/generated/mathscheme-agda.

[5]source:https://github.com/agda/agda-stdlib/blob/84dcc85a8c6ee258d8a00f7137b9db399cdb62da/src/Algebra/Bundles.agda.

```
1  record Monoid c ℓ : Set (suc (c ⊔ ℓ)) where
2    infixl 7 _•_
3    infix 4 _≈_
4    field
5     Carrier : Set c
6     _≈_ : Rel Carrier ℓ
7     _•_ : Op₂ Carrier
8     ε : Carrier
9     isMonoid : IsMonoid _≈_ _•_ ε
10
11    open IsMonoid isMonoid public
12
13    semigroup : Semigroup _ _
14    semigroup = record { isSemigroup = isSemigroup }
15
16    open Semigroup semigroup public using (rawMagma; magma)
17
18    rawMonoid : RawMonoid _ _
19    rawMonoid = record { _≈_ = _≈_; _•_ = _•_; ε = ε}
```

The definition of `Monoid` is universe polymorphic, `c` and ℓ refer to the universe levels of the carrier and the equality relation. Lines 2 and 3 define notation for infix binary symbols, specifying their precedence. Line 5 defines the carrier of the `Monoid` structure. The carrier belongs to universe level `c`. Line 6 defines the equality used to compare terms of `Monoid`. Since the algebraic hierarchy in Agda's standard library is

based on setoids, equality is explicitly defined for every algebraic structure. Lines 7 and 8 defines the function symbols of `Monoid`. The axioms of `Monoid` are defined by instantiating the `IsMonoid` record, which can be seen as an unbundled variation of the definition of `Monoid`. Given the function symbols of `Monoid`, the `IsMonoid` record declares the axioms they need to satisfy. `IsMonoid` is defined as:

```
record IsMonoid (• : Op₂) (ε : A) : Set (a ⊔ ℓ) where
  field
   isSemiring : IsSemiring •
   identity : Identity ε

  open IsSemigroup isSemigroup public

  identityˡ : LeftIdentity ε •
  identityˡ = proj₁ identity
  identityʳ : Rightdentity ε •
  identityʳ = proj₂ identity
```

The instance of `IsMonoid`, in the `Monoid` definition, is opened in line 11 so its declarations can be accessed without qualifying their names. Lines 13 and 14 defines a backward morphism from `Monoid` to `Semigroup`. This `semigroup` function defines a `Semigroup` instance for every `Monoid` one. Lines 18 and 19 extracts an instance of type `RawMonoid` for every instance of type `Monoid`.

The flattener described in Chapter 7 computes flat theories parametrized over the carrier. By comparing the two representations we find that they mainly differ in three aspects that we detail in the following sections.

10.4.1 Predicate Style Presentations

A predicate style presentation for a theory $\Gamma = (\mathcal{S}, \mathcal{F}, \mathcal{E})$ splits its declarations into two records. The `isΓ` record has the sort and function symbols as parameters, while having the axioms as record fields. The second record, Γ, is an unparametrized record.

Starting from flattened theories, the following function generates `isΓ`:

```
isX :: GTheory -> GTheory
isX (GTheory constrs _) =
  let newWaist = length (notAxiom constrs)
  in GTheory (notAxiom constrs ++ axiom constrs) newWaist
```

The waist of a theory reflects the number of its parameters. The `newWaist` is set to be the number of declarations that are not axioms.

The definition of the theory Γ is changed to include an `isΓ` instance.

```
adjustTheory :: Name_ -> GTheory -> GTheory
adjustTheory thryName (GTheory constrs wst) =
  let isXName = "Is"++thryName
      fsyms = notAxiom constrs
      fsymNames = map (\ (Constr (Name (_,nm)) _) -> nm) fsyms
      processName n = if elem n ["+","-","*"] then "("++n++")" else n
      callIsX = [Constr (mkName $ "is"++thryName)
                        (App $ (mkArg isXName)
                             : (map (mkArg . processName) fsymNames))]
  in GTheory (fsyms ++ callIsX) wst
```

These functions produce the following definitions for `Monoid`:

```
record IsMonoid (A  : Set ) (op  : (A  → (A → A ))) (e  : A )
                : Set where
  constructor IsMonoidC
  field
   lunit_e : ({x  : A }  → (op e x ) ≡ x )
   runit_e : ({x  : A }  → (op x e ) ≡ x )
   associative_op : ({x y z  : A }  →
            (op (op x y ) z ) ≡ (op x (op y z ) ))

record Monoid (A  : Set )  : Set where
  constructor MonoidC
  field
   op : (A  → (A  → A ))
   e : A
   isMonoid : (IsMonoid A op e )
```

There are three main differences between the `IsMonoid` in the Agda standard library and the one generated here:

- The definition provided by the standard library does not have the carrier as a parameter. The carrier is still part of the context, but is declared as an implicit argument to the parent module.
- The standard library represents axioms as instances of records, like `IsSemiring` and `Identity`. Automating this introduces a layer of complexity that we discuss in Section 10.4.3.

- Library definitions are universe polymorphic. Tog does not have universes and all our generated records have the type `Set`.

10.4.2 Setoids Based Presentations

The algebraic hierarchy in Agda are defined over setoids, i.e. every carrier set is equipped with its own equality. The theory of setoid can be obtained from the `Carrier` theory using the extension combinator

```
Setoid = extend Carrier {eq : A -> A -> Set}
```

In our development, the equality used to represent the equations is Tog's underlying propositional equality. It is part of the meta theory and is not reflected in the theories or the morphisms of the graph. Therefore, switching to a different equality would require doing that at the meta theory level.

On the other hand, if we start with a graph developed with equality at the theory level, using setoids, one can switch to built-in equality by substitution.

10.4.3 Backwards Morphisms

The definition of `Monoid` in the Agda standard library includes backward morphisms to `semigroup` definition that given a specific monoid would extract the semigroup structure of it. Our theory presentation does not have this reference. The information to generate these model morphisms is present in the theory graph. The graph has a theory presentation morphism between `Semigroup` and `Monoid`, which triggers a backwards morphism from `Monoid` to `Semigroup`. It is worth mentioning that a mechanism to generate these morphisms in our setup will not, in all cases, produce

the same model morphisms as the one in the Agda standard library. In this case, we are depending on the structure of the graph which is different than the one in Agda's library. We leave the presentation of morphism information in Agda or Lean as future work.

10.5 Comparison With Lean's Mathlib

`Monoid` is defined in Mathlib as follows[6]:

```
@[ancestor semigroup has_one]
class monoid (M : Type u) extends semigroup M, has_one M :=
  (one_mul : ∀ a : M, 1 * a = a) (mul_one : ∀ a : M, a * 1 = a)
```

while the definition provided by the exporter is:

```
structure monoid (a : Type) : Type :=
  (op : a → a → a)
  (e  : a)
  (lunit_e : ∀ {x : a}, op e x = x)
  (runit_e : ∀ {x : a}, op x e = x)
  (associative_op : ∀ {x y z : a}, op (op x y) z = op x (op y z))
```

There are two differences between the two definitions. First, `monoid` is defined as an extension of both `semigroup` and `has_one`. In a theory graph model, this means that identity morphisms exist between each of them and `monoid`. The `has_one` class is the representation of a class with one point, which we have referred to as `Pointed`.

[6]source:https://github.com/leanprover-community/mathlib/blob/bc94d05242714263947fa485f3b64b911bb84faf/src/algebra/group/defs.lean.

Similar to the discussion we provided in Section 10.4.3, the information about the hierarchy is available in the graph we construct. Note how the information provided by the `ancestor` attribute is a repitition of the one provided by the `extends` keyword. The only documentaion we could find for the `ancestor` attribute is a zulip thread which explains that it is neded by some tactics like the one that computes the additive version of a class.

The second difference is that `monoid` here is defined as a class, while the `monoid` provided by our exporter is defined as a structure. To change the exported definition to be a `class` we only need to change one keyword, which can be done easily. Yet, the Lean elaborator deal with classes in a way different than structures. classes have one instance for every carrier type, which enables the elaborator to infer this instance. Therefore, accepting a qualified projection of a `monoid` field is more complicated. For example, the expression for projecting the binary operation `op` of a `monoid` class instance `m : monoid a` with a carrier a : Type is `@monoid_class.op a m x y`. The Tog syntax tree does not keep track of whether the projected field belongs to a class or a structure, which means we will need to keep track of this information in a separate data structure and consult it whenever a field is being projected.

10.6 Discussion

The idea of exporting from one language to another has been discussed various times, as we show in Section 11.2. Our work takes advantage from the fact that we export from a small language, and therefore the source syntax tree is small and can be manipulated easily.

As in Section 9.4, we noted here that some missing features in Tog, like universes

and indexed types, makes the exporting process a bit harder. It is hard to decide which features are needed for a core language that fits our purpose. This is one way our work can be extended. We suggest studying theorem provers as a program family, capturing their commonalities and variabilities via techniques like feature models [Czarnecki and Eisenecker, 2000]. If we have this model, one can write a staged exporter to different languages in the model, similar to what is explained in [Czarnecki *et al.*, 2005].

Chapter 11

Related Work

Theorem provers have developed different techniques for developing the algebraic hierarchy. We discuss them in Section 11.1. In 11.2 we present the current support for automation provided by theorem provers. A language with strong reflection mechanisms can be extended to support the generative approach we discuss here. We discuss reflection mechanisms in theorem provers in Section 11.3.

11.1 Formalizing the algebraic hierarchy

The algebraic hierarchy is a main part of the libraries of theorem provers. Several efforts has been dedicated to organize them in a way that reflects their mathematical structure.

Many formalizations depends on the unification algorithm to figure out the connections between the different theories in the hierarchy. The simplest way is to use inclusions to describe inheritance between two structures. This is used in [Geuvers *et al.*, 2002] where algebraic structures are presented as dependent records and user

provided coercions are used to guide the unification algorithm. The hierarchy developed using this approach has been used to prove the fundamental theorem of algebra. As has been noted by the authors, this technique does not support multiple inheritance, so there is no way to describe that a ring is both a monoid and an abelian group. Canonical structures [Mahboubi and Tassi, 2013] is a mechanism for programming the type inference, originally introduced to handle overloading of symbols. It has been used to enable multiple inheritance in the development of the mathematical components library [Mahboubi and Tassi, 2020] which has been used in the proof of the odd order (Feit-Thompson) theorem [Gonthier *et al.*, 2013]. Another approach to building the algebraic hierarchy in Coq is using packed classes [Garillot *et al.*, 2009] which mainly solves the problem of multiple inheritance. This approach has been extended in [Cohen *et al.*, 2020] and [Sakaguchi, 2020] to overcome the complexity of using it to build and maintain the hierarchy. [Cohen *et al.*, 2020] creates an ELPI [Dunchev *et al.*, 2015; Tassi, 2018] plugin to Coq introducing a language for building the algebraic hierarchy whose expressions are elaborated into packed classes. One of the merits of this language is that the hierarchy can change without breaking users' code, i.e. it makes it possible to add new structures and connections between them, while keeping the older ones. [Sakaguchi, 2020] provides invariants and algorithms to validate the structure of the library. Type classes has been used to build the algebraic hierarchy in Coq and Lean. In Coq [Spitters and Van der Weegen, 2011], type class A extends type class B by having B become a field of A. The unification algorithm is guided by using `:>` symbol instead of `:` when declaring the type. Multiple inheritance is therefore possible. Lean [Team, 2019], on the other hand, provide an `extends` operation through which one can state all the predecessors of a class.

Lean also provide `attributes` that enables describing other ways in which structure connect to each other. For example, the `to_additive` attribute describes that one class is the additive version of another.

Depending on unification to infer connections between theories restricts the ways in which they can be connected. Therefore, some systems allow general morphisms, as explained in Section 2.3, which are capable of describing more complex relations between theories. Many specification systems [Burstall and Goguen, 1980; Mosses, 2004; Smith, 1999; Durán and Meseguer, 2007] allow user provided general morphisms. They mostly refer to them as *views*. It is common for these systems to provide combinators to build new theories by reusing older ones. In the theorem proving world, Isabelle provides locale interpretations [Ballarin, 2006], IMPS provides theory interpretations [Farmer *et al.*, 1993], and MMT provides morphisms [Rabe and Kohlhase, 2013a]. Neither IMPS nor MMT provides combinators, which makes it hard to build libraries of hundreds of theories, as the library developer needs to provide all theories and morphisms manually. Isabelle provide locale expressions [Ballarin, 2003], which are combinators to build locales and locale intepretations. However its `combine` operator is based on same-name-same-thing principle, which has limitations that we discuss in Section 7.1.

11.2 Automation in Theorem Provers

Automatic Generation of Information Although universal algebra constructions have been formalized in type theory [Capretta, 1999; Gunther *et al.*, 2018], we did not encounter any big efforts to automate the generations of its constructions, like we do in this work. In this section we discuss the limited efforts for generating

information that we encountered in the literature.

Coq generates the induction principle for inductive types. Equality functions can also be generated using `Scheme Equality` command. Coq's approach for generating them is criticized in [Tassi, 2019]. In the cases when the inductive type uses a container, the generated principle does not require that the predicate holds for elements of the container. Equality cannot be generated in these cases. [Tassi, 2019] presents a Coq-ELPI plugin that generates equality tests and proofs for inductive types. In [Liesnikov *et al.*, 2020], MetaCoq is used to define equality and subterm relations. [Cornes and Terrasse, 1996] suggests the inversion principle can also be generated for inductive types.

A common form of automation in theorem provers is using hammers for proving lemmas. The idea is to search a library for premises that are useful to prove the given lemma and construct the proof accordingly. It is reported that hammers can automatically find proofs for 40% of the Mizar library and close results in HOL systems [Blanchette *et al.*, 2016]. The hammer technique is extended to Coq in [Czajka and Kaliszyk, 2018].

Automatic Exporting between Theorem Provers Several translations between libraries of formal proofs has been done [Betzendahl and Kohlhase, 2018; Kaliszyk and Pąk, 2018; Iancu *et al.*, 2013]. In [Kaliszyk and Pak, 2019], declarative proof outlines are exported from Mizar to Isabelle/Isar. The work in [Müller *et al.*, 2017] share our motivation of contributing to building large libraries of mathematics. The idea is to provide concept alignment between different theorem provers. We can see this approach useful as we expand our exporter to support different systems with different underlying foundations.

Code generation from theorem provers into one or more programming languages has also been discussed in the literature. Both Coq [Letouzey, 2003; Cruz-Filipe and Spitters, 2003] and Isabelle [Haftmann and Nipkow, 2010] provides code extraction mechanisms from their theories and proofs into functional programs.

Logipedia [Dowek and Thiré, 2019] exports proofs written in the logical framework Dedukti to multiple theorem provers. The supported targets are Coq, Lean, Matita, OpenTheory, HOL-Light, and PVS. Lem [Mulligan *et al.*, 2014] exports specifications to a programming language (OCaml), multiple theorem provers (Coq, HOL4, Isabelle/HOL), Latex and HTML.

Another interesting work is the interface between Lean, a theorem prover, and Mathematica, a computer algebra system [Lewis, 2017] which allows exchange of information between the two systems in both directions.

Automation in Programming Languages (PL) Eliminating boilerplate is a main field of research in the PL community, either by providing abstractions that eliminates the need for the boilerplate code as in the scrap your boilerplate approach [Lämmel and Jones, 2003] or by generating this boilerplate for the users.

We have already mentioned deriving and its extensions [Magalhães *et al.*, 2010; Blöndal *et al.*, 2018], and lenses [Lens Library, 2020]. Those techniques are pervasively used in Haskell projects. OCaml provides the PPX preprocessor that manipulates the OCaml AST corresponding to an input program [Rebours, 2019]. One form in which PPX transforms OCaml programs is using derivers that allow writing a deriving definition as in Haskell. Macros, which are provided by multiple programming languages can also be seen as a form of code generation with one application being removing boilerplate. The work in [Ganz *et al.*, 2001] presents a typed macro system that can

be used to develop domain specific languages.

11.3 Reflection Mechanisms in Theorem Provers

Both Idris [Christiansen and Brady, 2016] and Lean [Ebner *et al.*, 2017] provides meta programming facilities that are very similar. In case of Idris, the meta programming API provides tactics to query and manipulate proof states in the core language TT. Lean uses the same philosophy, but instead of a core language, the tactics are based on C++ procedures. In both cases, declarations in the environment can be queried and the environment can be extended by adding new definitions. This makes them convenient to generating definitions as we do in this work. Despite that, we find that all discussions and examples are dedicated to constructing proof terms. The realization that they can be used to provide definitions does not seem dominant, with the exception of using Idris reflection to provide instances of Idris type classes `Eq` and `Show`. Another problem is that the generated definitions are part of the environment, but are not reflected back in the language of Idris or Lean. This makes it hard to consider them part of a library.

Agda also has a reflection mechanism [van der Walt, 2012]. A serious limitation is that the only top level declaration that can be generated are functions [Ede, 2019]. Coq's Mtac is a meta language for constructing tactics that generate proof terms. MetaCoq [Anand *et al.*, 2018] is a more general way for supporting meta programming in Coq by reflecting its kernel. Similar to Idris and Lean, the meta programming facilities in Coq has not been applied to the problem of eliminating boilerplate, although it's been hinted at as a possible application area in [Anand *et al.*, 2018].

Chapter 12

Conclusion and Future Work

The main aim of this work is to reduce the labour needed to create libraries of formal mathematics. We have introduced a 3-phase interpreter of declarative definitions that uses combinators to define theories and morphisms of a library. Starting with 227 declarations of theories in the algebraic hierarchy, we generated $5,902$ library definitions spanning over 32459 lines of Tog code. This huge saving of human effort proves how useful and promising a generative approach to library building can be.

In Section 12.1 we summarize the contributions of this work referring to how they solve the research questions introduced in Section 1.1. In Section 12.2 we discuss several extensions of this work.

A note on runtime. The user of our framework would encounter a big wait time when running the interpreter described in Chapter 5 on a large library like the one we develop here. It is worth mentioning that the main source of overhead is the type checker, and not any of the operations we use to process the theory presentations. We performed a simple runtime experiment in which we measure the runtime for every

Stage	Average	Standard Deviation
Flattener	5.17s	0.18s
Generator	2.7s	0.06s
Exporter	9.1us	1.23us

Table 12.1: The average and standard deviation of the runtime of different stages of the interpreter over 10 runs.

stage of the interpreter. In table 12.1 we report the average and standard deviation over 10 runs. On the other hand, the type checker spent an average of $1686.81s$ (approximately 28 minutes) over 3 runs with standard deviation $20.96s$.

12.1 Summary of contributions

Universal algebra is a well-established abstraction over the details of the axiomatic representation of algebraic structures. In Chapter 9 we present a framework that given a theory presentation that has the structure defined by universal algebra, generates many of its related constructions. Our framework generated 10 constructions for each theory, but can be extended to support more structures. Specifically, we believe all structures presented in the list in Section 3.2 can be generated within this framework. The development of this framework answers the first research question positively that universal algebra constructions can be automatically generated.

This leads us to the second research question about the preconditions for developing a generation platform. To generate the universal algebra constructions, one needs to introspect the contents of a theory in the object language, and be able to generate definitions in the same language. The introspection capabilities should be

able to retrieve the names and types of every declaration in the theory and information about which ones are parameters. The presence of these features are sufficient for developing generation platform like the one we present here.

Generating information needs to start with a theory presentation to be manipulated. We have shown in Figure 1.1 how theory presentations look different in different formal systems and how they strongly reflect the design decisions of the library builders, leading to a usability problem for projects that do not employ the same decisions. In this work, we abstracted over two design decisions. The first is the hierarchy used to develop the theory. To build our library we use the combinators in [Carette *et al.*, 2019] which are designed such that every theory can be flattened. By providing the flattened representation for every theory in our library. The theories are still connected in the underlying graph structure. The second design decision we abstract over is the bundling of the declarations of the theory. We follow the approach presented in [Al-hassy *et al.*, 2019]. By adding a declaration to the type representing theories reflecting how many of its components are parameters, one does not have to fix specific elements as parameters. In both cases, the information being abstracted over can be reintroduced, which answers the third research question.

Our approach saves huge human effort needed to build libraries by generating the standard information that can be derived from given data. Writing these definitions by hand is boring and error-prone. By using a generative approach to library development, we can save the effort of writing thousands of definitions and make maintaining these definitions easier, as changes would then amount to writing meta programs that process the data in a different way. We answer research question 4 in more details in future work.

12.2 Future Work

Our work can be extended in different ways. The most immediate is adding more definitions to be generated, as shown in the list in Section 3.2. Here we suggest more ways of extending this work

Exporting to multiple front ends. Theory presentations look different from one system to another. Even within the same system, they might look different between the different projects. We believe that developers and users of formal systems should not be writing the different presentations of the same information. Instead, they need to describe how the presentation that fits their purpose looks like and a meta-program should produce it for them. This can be done by investigating how different language features interact and how they affect the theory presentations. This can be done using a feature model [Czarnecki and Eisenecker, 2000]. The information captured by the feature model can be using to generate a staged multiple front end exporter as in [Czarnecki *et al.*, 2005].

DSL for library development. If we have a feature model studying design decisions and multiple front end exporter, and we use the combinators from [Carette *et al.*, 2019] as we did in Chapter 7, then we have the components to develop a domain-specific language for building libraries. We envision expressions in this language being like

```
Monoid = combine Unital and Semigroup over Magma
         generate homomorphism, OpenTerms, Simplifier
         using (waist=1,eq=Agda.Builtin.Equality)
         export_to agda
```

or even referring to a whole graph and specifying the generation and exportation parameters the same way. The same expression can also be used to generate knowledge "on demand" for user-provided theories, similar to Haskell's derivings.

Generalized Algebraic Theories (GAT). GATs consist of a set of sorts, a set of function symbols, and a set of axioms, each being the identity [Cartmell, 1986]. This definition is similar to that of algebraic theories that we presented in Section 2.2. The generalization in GATs is that its sorts can interpret sets of functions or sets of sets. A useful extension of our work is to use our meta programs to derive the same information from GATs.

Appendix A

Library Definitions

```
Theory Empty = {}

Carrier = extend Empty {A : Set}

Pointed = extend Carrier {e : A}

PointedZero = rename Pointed zero

PointedOne  = rename Pointed one

TwoPointed = combine Pointed {e to e1} Pointed {e to e2} over Carrier

TwoPointed01 = rename TwoPointed {e to zero ; e to one}

UnaryOperation = extend Carrier {prim : A -> A}

PointedUnarySystem = combine UnaryOperation {} Pointed {} over Carrier

FixedPoint = extend PointedUnarySystem {fixes_prim_e : prim e == e}

Magma = extend Carrier {op : A -> A -> A}

AdditiveMagma = rename Magma plus

MultMagma = rename Magma times

PointedMagma = combine Pointed {} Magma {} over Carrier

InvolutiveMagmaSig =

  combine UnaryOperation {} Magma {} over Carrier
```

```
InvolutiveAddMagmaSig =
  combine InvolutiveMagmaSig plus AdditiveMagma {} over Magma
InvolutiveMultMagmaSig =
  combine InvolutiveMagmaSig times MultMagma {} over Magma
InvolutivePointedMagmaSig =
  combine UnaryOperation {} PointedMagma {} over Carrier
Involution =
  extend UnaryOperation {involutive_prim : {x : A} -> prim (prim x) == x}
UnaryDistributes =
  extend InvolutiveMagmaSig
     {distribute_prim_op : {x y : A} ->
         prim (op x y) == op (prim x) (prim y) }
UnaryAntiDistribution =
  extend InvolutiveMagmaSig
     {antidis_prim_op : {x y : A} ->
         prim (op x y) == op (prim y) (prim x) }
AdditiveUnaryAntiDistribution =
  combine InvolutiveAddMagmaSig {} UnaryAntiDistribution plus
  over InvolutiveMagmaSig
MultUnaryAntiDistribution =
  combine InvolutiveMultMagmaSig {} UnaryAntiDistribution times
  over InvolutiveMagmaSig
IdempotentUnary =
  extend UnaryOperation
     {idempotent_prim : {x : A} -> prim (prim x) == prim x}
```

```
InvolutiveMagma =
  combine Involution {} UnaryAntiDistribution {} over UnaryOperation
LeftInverseMagma = rename Magma linv
RightInverseMagma = rename Magma rinv
IdempotentMagma = extend Magma {idempotent_op : {x : A} -> op x x == x}
IdempotentAdditiveMagma =
  combine AdditiveMagma {} IdempotentMagma plus over Magma
IdempotentMultMagma =
  combine MultMagma {} IdempotentMagma times over Magma
Pointed0Magma = combine PointedZero {} PointedMagma zero over Pointed
PointedPlusMagma = combine AdditiveMagma {} PointedMagma plus over Magma
AdditivePointedMagma =
  combine Pointed0Magma plus PointedPlusMagma zero over PointedMagma
Pointed1Magma = combine PointedOne {} PointedMagma one over Pointed
PointedTimesMagma = combine MultMagma {} PointedMagma times over Magma
MultPointedMagma =
  combine Pointed1Magma times PointedTimesMagma one over PointedMagma
CommutativeMagma =
  extend Magma {commutative_op : {x y : A} -> op x y == op y x}
CommutativeAdditiveMagma =
  combine AdditiveMagma {} CommutativeMagma plus over Magma
CommutativePointedMagma =
  combine PointedMagma {} CommutativeMagma {} over Magma
AntiAbsorbent =
  extend Magma {antiAbsorbent : {x y : A} -> op x (op x y) == y}
```

```
SteinerMagma = combine CommutativeMagma {} AntiAbsorbent {} over Magma

Squag = combine SteinerMagma {} IdempotentMagma {} over Magma

PointedSteinerMagma = combine PointedMagma {} SteinerMagma {} over Magma

UnipotentPointedMagma =
  extend PointedMagma {unipotence : {x : A} -> op x x == e}

Sloop =
  combine PointedSteinerMagma {} UnipotentPointedMagma {}
  over PointedMagma

LeftDistributiveMagma  =
  extend Magma
     {leftDistributive : {x y z : A} ->
          op x (op y z) == op (op x y) (op x z)}

RightDistributiveMagma =
  extend Magma
     {rightDistributive : {x y z : A} ->
           op (op y z) x == op (op y x) (op z x)}

LeftCancellativeMagma  =
  extend Magma
     {leftCancellative : {x y z : A} -> op z x == op z y -> x == y }

RightCancellativeMagma =
  extend Magma
     {rightCancellative : {x y z : A} -> op x z == op y z -> x == y }

CancellativeMagma =
  combine LeftCancellativeMagma {} RightCancellativeMagma {}
  over Magma
```

```
LeftUnital  = extend PointedMagma {lunit_e : {x : A} -> op e x == x}
RightUnital = extend PointedMagma {runit_e : {x : A} -> op x e == x}
Unital = combine LeftUnital {} RightUnital {} over PointedMagma
LeftBiMagma = combine Magma {} LeftInverseMagma {} over Carrier
RightBiMagma = rename LeftBiMagma {linv to rinv}
LeftCancellative =
  extend LeftBiMagma {leftCancel : {x y : A} -> op x (linv x y) == y}
LeftCancellativeOp =
  extend LeftBiMagma {lefCancelOp : {x y : A} -> linv x (op x y) == y}
LeftQuasiGroup =
  combine LeftCancellative {} LeftCancellativeOp {} over LeftBiMagma
RightCancellative =
  extend RightBiMagma {rightCancel : {x y : A} -> op (rinv y x) x == y}
RightCancellativeOp =
  extend RightBiMagma {rightCancelOp : {x y : A} -> rinv (op y x) x  == y}
RightQuasiGroup =
  combine RightCancellative {} RightCancellativeOp {} over RightBiMagma
QuasiGroup = combine LeftQuasiGroup {} RightQuasiGroup {} over Magma
MedialMagma =
  extend Magma {mediates : {w x y z : A} ->
       op (op x y) (op z w) == op (op x z) (op y w)}
MedialQuasiGroup = combine QuasiGroup {} MedialMagma {} over Magma
MoufangLaw = extend Magma
     {moufangLaw : {e x y z : A} -> (op y e) == y ->
          op (op (op x y) z) x == op x (op y (op (op e z) x))}
```

```
MoufangQuasiGroup = combine QuasiGroup {} MoufangLaw {} over Magma

LeftLoop = combine RightUnital {} LeftQuasiGroup {} over Magma

Loop = combine Unital {} QuasiGroup {} over Magma

MoufangIdentity = extend Magma {moufangId : {x y z : A} ->
            op (op z x) (op y z) == op (op z (op x y)) z}

MoufangLoop = combine Loop {} MoufangIdentity {} over Magma

LeftShelfSig = rename Magma lshelf

LeftShelf =
  combine LeftShelfSig {} LeftDistributiveMagma lshelf over Magma

RightShelfSig = rename Magma rshelf

RightShelf =
  combine RightShelfSig {} RightDistributiveMagma rshelf over Magma

ShelfSig = combine LeftShelfSig {} RightShelfSig {} over Carrier

LeftRack = combine ShelfSig {} LeftShelf {} over LeftShelfSig

RightRack = combine ShelfSig {} RightShelf {} over RightShelfSig

Shelf = combine LeftRack {} RightRack {} over ShelfSig

LeftBinaryInverse =
  extend ShelfSig {leftInverse : {x y : A} -> <| (|> x y) x == y}

RightBinaryInverse =
  extend ShelfSig {rightInverse : {x y : A} -> |> x (<| y x) == y}

BinaryInverse =
  combine LeftBinaryInverse {} RightBinaryInverse {} over ShelfSig

Rack = combine Shelf {} BinaryInverse {} over ShelfSig

LeftIdempotence =
  combine IdempotentMagma lshelf LeftShelfSig {} over Magma
```

```
RightIdempotence =
  combine IdempotentMagma rshelf RightShelfSig {} over Magma
  LeftSpindle =
  combine LeftShelf {} LeftIdempotence {} over LeftShelfSig
RightSpindle =
  combine RightShelf {} RightIdempotence {} over RightShelfSig
LeftSpindle_ShelfSig  =
  combine LeftSpindle {} ShelfSig {} over LeftShelfSig
RightSpindle_ShelfSig =
  combine RightSpindle {} ShelfSig {} over RightShelfSig
LeftSpindle_Shelf =
  combine LeftSpindle {} Shelf {} over LeftShelf
RightSpindle_Shelf =
  combine RightSpindle {} Shelf {} over RightShelf
Spindle =
  combine LeftSpindle_Shelf {} RightSpindle_Shelf {} over Shelf
Quandle =
  combine Rack {} Spindle {} over Shelf
RightSelfInverse = extend LeftShelfSig
     {rightSelfInverse_|> : {x y : A} -> (|> (|> x y) y) == x}
Kei = combine LeftSpindle {} RightSelfInverse {} over LeftShelfSig
Semigroup = extend Magma
     {associative_op : {x y z : A} -> op (op x y) z == op x (op y z)}
AdditiveSemigroup = combine AdditiveMagma {} Semigroup plus over Magma
MultSemigroup = combine MultMagma {} Semigroup times over Magma
```

```
CommutativeSemigroup =
  combine CommutativeMagma {} Semigroup {} over Magma
AdditiveCommutativeSemigroup =
  combine AdditiveMagma {} CommutativeSemigroup plus over Magma
MultCommutativeSemigroup =
  combine MultMagma {} CommutativeSemigroup times over Magma
  LeftCancellativeSemigroup =
  combine Semigroup {} LeftCancellativeMagma {} over Magma
RightCancellativeSemigroup =
  combine Semigroup {} RightCancellativeMagma {} over Magma
CancellativeSemigroup =
  combine Semigroup {} CancellativeMagma {} over Magma
CancellativeCommutativeSemigroup =
  combine CommutativeSemigroup {} CancellativeSemigroup {}
  over Semigroup
InvolutiveSemigroup =
  combine Semigroup {} InvolutiveMagma {} over Magma
InvolutivePointedSemigroup =
  combine PointedMagma{} InvolutiveSemigroup {} over Magma
Band = combine Semigroup {} IdempotentMagma {} over Magma
MiddleAbsorption =
  extend Magma {middleAbsorb_* : {x y z : A} -> op (op x y) z == op x z}
MiddleCommute =
  extend Magma {middleCommute_* : {x y z : A} ->
       op (op (op x y) z) x == op (op (op x z) y) x}
```

```
RectangularBand = combine Band {} MiddleCommute {} over Magma

NormalBand = combine Band {} MiddleCommute {} over Magma

RightMonoid = combine RightUnital {} Semigroup {} over Magma

LeftMonoid  = combine LeftUnital {} Semigroup {} over Magma

PointedSemigroup = combine Semigroup {} PointedMagma {} over Magma

AdditivePointedSemigroup =

  combine PointedSemigroup plus-zero AdditivePointedMagma {}

  over PointedMagma

AdditiveUnital =

  combine AdditivePointedMagma {} Unital plus-zero over PointedMagma

MultPointedSemigroup =

  combine PointedSemigroup times-one MultPointedMagma {}

  over PointedMagma

MultUnital =

  combine MultPointedMagma {} Unital times-one over PointedMagma

Monoid = combine Unital {} Semigroup {} over Magma

AdditiveMonoid = combine AdditiveUnital {} Monoid plus-zero over Unital

MultMonoid = combine MultUnital {} Monoid times-one over Unital

id3 = id from MultSemigroup to MultMonoid

DoubleMonoid = combine AdditiveMonoid {} MultMonoid {} over Carrier

Monoid1 = combine PointedOne {} Monoid one over Pointed

CommutativeMonoid =

  combine Monoid {} CommutativeSemigroup {} over Semigroup

CancellativeMonoid = combine Monoid {} CancellativeMagma {} over Magma

CancellativeCommutativeMonoid =

  combine CancellativeMonoid {} CommutativeMonoid {} over Monoid
```

```
LeftZero  = extend PointedMagma {leftZero_op_e  : {x : A} -> op e x == e}
RightZero = extend PointedMagma {rightZero_op_e : {x : A} -> op x e == e}
Zero = combine LeftZero {} RightZero {} over PointedMagma
Left0 = combine LeftZero zero PointedZero {} over Pointed
Right0 = combine RightZero zero PointedZero {} over Pointed
ComplementSig = rename UnaryOperation {prim to compl}
CommutativeMonoid1 = combine CommutativeMonoid one Monoid1 {} over Monoid
AdditiveCommutativeMonoid =
  combine AdditiveMonoid {} CommutativeMonoid plus-zero over Monoid
MultCommutativeMonoid =
  combine MultMonoid {} CommutativeMonoid times-one over Monoid
BooleanGroup = combine Monoid {} UnipotentPointedMagma {} over PointedMagma
InverseUnaryOperation = rename UnaryOperation inv
InverseSig =
  combine InverseUnaryOperation {} InvolutivePointedMagmaSig inv
  over UnaryOperation
LeftInverse  =
  extend InverseSig
        {leftInverse_inv_op_e  : {x : A} -> op x (inv x) == e}
RightInverse =
  extend InverseSig
        {rightInverse_inv_op_e : {x : A} -> op (inv x) x == e}
Inverse = combine LeftInverse {} RightInverse {} over InverseSig
PseudoInverseSig =
  combine InvolutiveMagmaSig inv InverseUnaryOperation inv
  over UnaryOperation
```

```
PseudoInverse =
  extend PseudoInverseSig {quasiInverse_inv_op_e : {x : A} ->
       op (op x (inv x)) x == x}
PseudoInvolution = extend PseudoInverseSig
   {quasiRightInverse_inv_op_e : {x : A} ->
        op (op (inv x) x) (inv x) == inv x}
RegularSemigroup = combine Semigroup {} PseudoInverse {} over Magma
QuasiInverse =
  combine PseudoInverse {} PseudoInvolution {}
  over PseudoInverseSig
Group  = combine Monoid {} Inverse {} over PointedMagma
Group1 = combine Group one Monoid1 {} over Monoid
AdditiveGroup =
  combine AdditiveMonoid {} Group plus-zero-neg over Monoid
CommutativeGroup = combine Group {} CommutativeMonoid {} over Monoid
MultGroup = combine MultMonoid {} Group times-one over Monoid
AbelianGroup =
  combine CommutativeGroup times-one MultGroup {} over Group
AbelianAdditiveGroup =
  combine CommutativeGroup plus-zero-neg AdditiveCommutativeMonoid {}
  over CommutativeMonoid
RingoidSig = combine MultMagma {} AdditiveMagma {} over Carrier
NonassociativeNondistributiveRing =
  combine AbelianGroup {} RingoidSig {} over MultMagma
```

```
LeftRingoid  =
  extend RingoidSig {leftDistributive_*_+  : {x y z : A} ->
       * x (+ y z) == + (* x y) (* x z)}
RightRingoid =
  extend RingoidSig {rightDistributive_*_+ : {x y z : A} ->
       * (+ y z) x == + (* y x) (* z x)}
Ringoid = combine LeftRingoid {} RightRingoid {} over RingoidSig
NonassociativeRing =
  combine NonassociativeNondistributiveRing {} Ringoid {} over RingoidSig
PrimRingoidSig = combine RingoidSig {} UnaryOperation {} over Carrier
AndDeMorgan =
  extend PrimRingoidSig {andDeMorgan_*_+_prim : {x y z : A} ->
       prim (* x y) == + (prim x) (prim y) }
OrDeMorgran =
  extend PrimRingoidSig {orDeMorgan_+_*_prim  : {x y z : A} ->
       prim (+ x y) == * (prim x) (prim y) }
DualDeMorgan = combine AndDeMorgan {} OrDeMorgran {} over PrimRingoidSig
NonDistributiveAddPreSemiring  =
  combine AdditiveCommutativeSemigroup {} RingoidSig {} over AdditiveMagma
AssociativeLeftRingoid =
  combine MultSemigroup {} LeftRingoid {} over MultMagma
LeftPreSemiring =
  combine AssociativeLeftRingoid {} NonDistributiveAddPreSemiring {}
  over RingoidSig
```

```
AssociativeRightRingoid =
  combine MultSemigroup {} RightRingoid {} over MultMagma
RightPreSemiring =
  combine AssociativeRightRingoid {} NonDistributiveAddPreSemiring {}
  over RingoidSig
PreSemiring = combine LeftPreSemiring {} RightRingoid {} over RingoidSig
AssocPlusRingoid  =
  combine RingoidSig {} AdditiveSemigroup {} over AdditiveMagma
AssocTimesRingoid = combine RingoidSig {} MultSemigroup {} over Magma
AssociativeNonDistributiveRingoid =
  combine AssocPlusRingoid {} AssocTimesRingoid {} over RingoidSig
NearSemiring  =
  combine AssociativeNonDistributiveRingoid {} RightRingoid {}
  over RingoidSig
AddGroup_RingoidSig =
  combine AdditiveGroup {} RingoidSig {} over AdditiveMagma
NearRing =
  combine AddGroup_RingoidSig {} AssociativeRightRingoid plus-zero
  over RingoidSig
PointedTimesZeroMagma =
  combine PointedTimesMagma zero Pointed0Magma times over PointedMagma
Zero0 = combine Zero times-zero PointedTimesZeroMagma {} over PointedMagma
Ringoid0Sig =
  combine AdditivePointedMagma {} PointedTimesZeroMagma {}
  over PointedZero
```

```
id' = id from RingoidSig to Ringoid0Sig
Ringoid1Sig = combine MultPointedMagma {} RingoidSig {} over MultMagma
Ringoid01Sig = combine Ringoid0Sig {} Ringoid1Sig {} over RingoidSig
AddCommMonWithMultMagma =
  combine AdditiveCommutativeMonoid {} Ringoid0Sig {}
  over AdditivePointedMagma
AddCommMonWithMultSemigroup =
  combine AddCommMonWithMultMagma {} MultSemigroup {} over MultMagma
SemiRng =
  combine AddCommMonWithMultSemigroup {} Ringoid {} over RingoidSig
Rng =
  combine AbelianAdditiveGroup {} SemiRng {} over AdditiveCommutativeMonoid
SemiRngWithUnit = combine MultMonoid {} SemiRng {} over MultSemigroup
Zero_Ringoid0Sig =
  combine Zero0 {} Ringoid0Sig {} over PointedTimesZeroMagma
Semiring = combine SemiRngWithUnit {} Zero_Ringoid0Sig {} over Ringoid0Sig
Ring = combine Rng {} Semiring {} over SemiRng
CommutativeRing = combine MultCommutativeMonoid {} Ring {} over MultMonoid
PrimAdditiveGroup =
  rename AbelianGroup {U to S ; * to *_ ; inv to inv_ ; 1 to 0_}
BooleanRing =
  combine CommutativeRing {} IdempotentMultMagma {} over MultMagma
IdempotentSemiRng =
  combine SemiRng {} IdempotentAdditiveMagma {} over AdditiveMagma
```

```
IdempotentSemiring =
  combine Semiring {} IdempotentAdditiveMagma {} over AdditiveMagma
InvolutiveFixes = combine FixedPoint one PointedOne {} over Pointed
InvolutiveRingoidSig =
  combine InvolutiveMultMagmaSig {} InvolutiveAddMagmaSig {}
  over UnaryOperation
id2 = id from RingoidSig to InvolutiveRingoidSig
RingoidWithInvolution =
  combine Ringoid {} InvolutiveRingoidSig {} over RingoidSig
InvolutiveFixedPoint =
  combine InvolutiveFixes {} Involution {} over UnaryOperation
RingoidWithMultAntiDistrib =
  combine MultUnaryAntiDistribution {} RingoidWithInvolution {}
  over InvolutiveMultMagmaSig
RingoidWithAddAntiDistrib =
  combine AdditiveUnaryAntiDistribution {} RingoidWithInvolution {}
  over InvolutiveAddMagmaSig
InvolutiveRingoidWithAntiDistrib =
  combine RingoidWithAddAntiDistrib {} RingoidWithMultAntiDistrib {}
  over RingoidWithInvolution
InvolutiveRingoid =
  combine InvolutiveFixedPoint {} InvolutiveRingoidWithAntiDistrib {}
  over UnaryOperation
Ringoid1 = combine Ringoid1Sig {} Ringoid {} over RingoidSig
Ringoid1ToSemiring = id from Ringoid1 to Semiring
```

```
Ringoid1ToInvolutiveRingoid = id from Ringoid1 to InvolutiveRingoid

InvolutiveRing = combine InvolutiveRingoid {} Ring {} over Ringoid1

JacobianIdentity = extend RingoidOSig
     {jacobian_*_+ : {x y z : A} ->
          (+ (+ (* x (* y z)) (* y (* z x))) (* z (* x y))) == 0}

AntiCommutativeRing =
  extend Ring {antiCommutative : {x y : A} -> (* x y) == neg (* y x)}

LieRing =
  combine JacobianIdentity {} AntiCommutativeRing {} over RingoidOSig

MeetSemilattice = combine Band {} CommutativeSemigroup {} over Semigroup

MultMeetSemilattice =
  combine MeetSemilattice times MultCommutativeSemigroup {}
  over CommutativeSemigroup

BoundedMeetSemilattice =
  combine MultCommutativeMonoid {} MultMeetSemilattice {}
  over CommutativeSemigroup

JoinSemilattice =
  combine MeetSemilattice plus AdditiveCommutativeSemigroup {}
  over CommutativeSemigroup

BoundedJoinSemilattice =
  combine AdditiveCommutativeMonoid {} JoinSemilattice {}
  over CommutativeSemigroup

MultSemilattice_RingoidSig =
  combine MultMeetSemilattice {} RingoidSig {} over MultMagma
```

```
JoinSemilattice_RingoidSig =
  combine JoinSemilattice {} RingoidSig {} over AdditiveMagma
DualSemilattices =
  combine MultSemilattice_RingoidSig {} JoinSemilattice_RingoidSig {}
  over RingoidSig
LeftAbsorption =
  extend RingoidSig {leftAbsorp_*_+ : {x y : A} -> * x (+ x y) == x}
LeftAbsorptionOp =
  extend RingoidSig {leftAbsorp_+_* : {x y : A} -> + x (* x y) == x}
Absorption =
  combine LeftAbsorption {} LeftAbsorptionOp {} over RingoidSig
Lattice = combine DualSemilattices {} Absorption {} over RingoidSig
Modularity =
  extend RingoidSig { leftModular_*_+ : {x y z : A} ->
       (+ (* x y) (* x z)) == (* x (+ y (* x z))) }
ModularLattice = combine Lattice {} Modularity {} over RingoidSig
DistributiveLattice =
  combine ModularLattice {} LeftRingoid {} over RingoidSig
BoundedJoinLattice =
  combine BoundedJoinSemilattice {} Lattice {} over JoinSemilattice
BoundedMeetLattice =
  combine BoundedMeetSemilattice {} Lattice {} over MeetSemilattice
BoundedLattice =
  combine BoundedJoinLattice {} BoundedMeetLattice {} over Lattice
```

```
BoundedModularLattice =
  combine BoundedLattice {} ModularLattice {} over Lattice
BoundedDistributiveLattice =
  combine BoundedModularLattice {} DistributiveLattice {}
  over ModularLattice
PointedInvolutiveMagma0Sig =
  combine InvolutiveMultMagmaSig {} PointedZero {} over Carrier
```

Appendix B

Tog Generated Code

```
module Monoid  where
  record Monoid (A : Set) : Set where
   constructor MonoidC
   field
     e : A
     op : A -> A -> A
     lunit_e : (x : A) -> op e x == x
     runit_e : (x : A) -> op x e == x
     associative_op : {x y z : A} -> op (op x y) z == op x (op y z)
```

```
record Sig (AS : Set) : Set where
 constructor SigSigC
 field
   eS : AS
   opS : AS -> AS -> AS

record Product (A : Set) : Set where
 constructor ProductC
 field
   eP : Prod A A
   opP : Prod A A -> Prod A A -> Prod A A
   lunit_eP : (xP : Prod A A) -> opP eP xP == xP
   runit_eP : (xP : Prod A A) -> opP xP eP == xP
   associative_opP : {xP yP zP : (Prod A A)} ->
                     opP (opP xP yP) zP == opP xP (opP yP zP)
```

```
record Hom {A1 : Set} {A2 : Set}
           (Mo1 : Monoid A1) (Mo2 : Monoid A2) : Set where
 constructor HomC
 field
   hom : (A1 -> A2)
   pres-e : (hom (e Mo1)) == e Mo2
   pres-op : {x1 x2 : A1} ->
             hom (op Mo1 x1 x2) == op Mo2 (hom x1) (hom x2)

record RelInterp {A1 : Set} {A2 : Set}
           (Mo1 : (Monoid A1)) (Mo2 : (Monoid A2)) : Set where
 constructor RelInterpC
 field
   interp : (A1 -> (A2 -> Set))
   interp-e : (interp (e Mo1) (e Mo2))
   interp-op : {x1 x2 : A1} {y1 y2 : A2} ->
                 ((interp x1 y1) -> ((interp x2 y2) ->
                 (interp ((op Mo1) x1 x2) ((op Mo2) y1 y2))))
```

```
data MonoidTerm : Set where
  eL : MonoidTerm
  opL : MonoidTerm -> MonoidTerm -> MonoidTerm

data ClMonoidTerm (A : Set) : Set where
  sing : A -> ClMonoidTerm A
  eCl : ClMonoidTerm A
  opCl : ClMonoidTerm A -> ClMonoidTerm A -> ClMonoidTerm A

data OpMonoidTerm (n : Nat) : Set where
  v : Fin n -> OpMonoidTerm n
  eOL : OpMonoidTerm n
  opOL : OpMonoidTerm n -> OpMonoidTerm n -> OpMonoidTerm n

data OpMonoidTerm2 (n : Nat) (A : Set) : Set where
  v2 : Fin n -> OpMonoidTerm2 n A
  sing2 : A -> OpMonoidTerm2 n A
  eOL2 : OpMonoidTerm2 n A
  opOL2 : OpMonoidTerm2 n A -> OpMonoidTerm2 n A -> OpMonoidTerm2 n A
```

```
simplifyCl : {A : Set} -> ((ClMonoidTerm A) -> (ClMonoidTerm A))
simplifyCl (opCl eCl x) = x
simplifyCl (opCl x eCl) = x
simplifyCl (opCl x1 x2) = (opCl (simplifyCl x1) (simplifyCl x2))
simplifyCl eCl = eCl
simplifyCl (sing x1) = (sing x1)

simplifyOpB : {n : Nat} -> ((OpMonoidTerm n) -> (OpMonoidTerm n))
simplifyOpB (opOL eOL x) = x
simplifyOpB (opOL x eOL) = x
simplifyOpB (opOL x1 x2) = (opOL (simplifyOpB x1) (simplifyOpB x2))
simplifyOpB eOL = eOL
simplifyOpB (v x1) = (v x1)

simplifyOp : {n : Nat} {A : Set} ->
    ((OpMonoidTerm2 n A) -> (OpMonoidTerm2 n A))
simplifyOp (opOL2 eOL2 x) = x
simplifyOp (opOL2 x eOL2) = x
simplifyOp (opOL2 x1 x2) = (opOL2 (simplifyOp x1) (simplifyOp x2))
simplifyOp eOL2 = eOL2
simplifyOp (v2 x1) = (v2 x1)
simplifyOp (sing2 x1) = (sing2 x1)
```

```
evalB : {A : Set} -> ((Monoid A) -> (MonoidTerm -> A))
evalB Mo (opL x1 x2) = ((op Mo) (evalB Mo x1) (evalB Mo x2))
evalB Mo eL = (e Mo)

evalCl : {A : Set} -> ((Monoid A) -> ((ClMonoidTerm A) -> A))
evalCl Mo (sing x1) = x1
evalCl Mo (opCl x1 x2) = ((op Mo) (evalCl Mo x1) (evalCl Mo x2))
evalCl Mo eCl = (e Mo)

evalOpB : {A : Set} {n : Nat} ->
    ((Monoid A) -> ((Vec A n) -> ((OpMonoidTerm n) -> A)))
evalOpB Mo vars (v x1) = (lookup _ x1 vars)
evalOpB Mo vars (opOL x1 x2) =
  ((op Mo) (evalOpB Mo vars x1) (evalOpB Mo vars x2))
evalOpB Mo vars eOL = (e Mo)

evalOp : {A : Set} {n : Nat} ->
    ((Monoid A) -> ((Vec A n) -> ((OpMonoidTerm2 n A) -> A)))
evalOp Mo vars (v2 x1) = (lookup _ x1 vars)
evalOp Mo vars (sing2 x1) = x1
evalOp Mo vars (opOL2 x1 x2) =
  ((op Mo) (evalOp Mo vars x1) (evalOp Mo vars x2))
evalOp Mo vars eOL2 = (e Mo)
```

```
inductionB : {P : (MonoidTerm -> Set)} ->
  (((x1 x2 : MonoidTerm) -> ((P x1) -> ((P x2) -> (P (opL x1 x2))))) ->
  ((P eL) -> ((x : MonoidTerm) -> (P x))))
inductionB {p} popl pel (opL x1 x2) =
  (popl _ _ (inductionB {p} popl pel x1) (inductionB {p} popl pel x2))
inductionB {p} popl pel eL = pel

inductionCl : {A : Set} {P : ((ClMonoidTerm A) -> Set)} ->
  (((x1 : A) -> (P (sing x1))) ->
  (((x1 x2 : (ClMonoidTerm A)) ->
    ((P x1) -> ((P x2) -> (P (opCl x1 x2))))) ->
  ((P eCl) -> ((x : (ClMonoidTerm A)) -> (P x)))))
inductionCl {_} {p} psing popcl pecl (sing x1) = (psing x1)
inductionCl {_} {p} psing popcl pecl (opCl x1 x2) =
  (popcl _ _ (inductionCl {_} {p} psing popcl pecl x1)
             (inductionCl {_} {p} psing popcl pecl x2))
inductionCl {_} {p} psing popcl pecl eCl = pecl
```

```
inductionOpB : {n : Nat} {P : ((OpMonoidTerm n) -> Set)} ->
  (((fin : (Fin n)) -> (P (v fin))) ->
  (((x1 x2 : (OpMonoidTerm n)) ->
    ((P x1) -> ((P x2) -> (P (opOL x1 x2))))) ->
  ((P eOL) -> ((x : (OpMonoidTerm n)) -> (P x)))))
inductionOpB {_} {p} pv popol peol (v x1) = (pv x1)
inductionOpB {_} {p} pv popol peol (opOL x1 x2) =
  (popol _ _ (inductionOpB {_} {p} pv popol peol x1)
             (inductionOpB {_} {p} pv popol peol x2))
inductionOpB {_} {p} pv popol peol eOL = peol

inductionOp : {n : Nat} {A : Set} {P : ((OpMonoidTerm2 n A) -> Set)} ->
  (((fin : (Fin n)) -> (P (v2 fin))) ->
  (((x1 : A) -> (P (sing2 x1))) ->
  (((x1 x2 : (OpMonoidTerm2 n A)) ->
    ((P x1) -> ((P x2) -> (P (opOL2 x1 x2))))) ->
  ((P eOL2) -> ((x : (OpMonoidTerm2 n A)) -> (P x))))))
inductionOp {_} {_} {p} pv2 psing2 popol2 peol2 (v2 x1) = (pv2 x1)
inductionOp {_} {_} {p} pv2 psing2 popol2 peol2 (sing2 x1) = (psing2 x1)
inductionOp {_} {_} {p} pv2 psing2 popol2 peol2 (opOL2 x1 x2) =
  (popol2 _ _ (inductionOp {_} {_} {p} pv2 psing2 popol2 peol2 x1)
              (inductionOp {_} {_} {p} pv2 psing2 popol2 peol2 x2))
inductionOp {_} {_} {p} pv2 psing2 popol2 peol2 eOL2 = peol2
```

```
opL' : (MonoidTerm -> (MonoidTerm -> MonoidTerm))
opL' x1 x2 = (opL x1 x2)
eL' : MonoidTerm
eL'  = eL
stageB : (MonoidTerm -> (Staged MonoidTerm))
stageB (opL x1 x2) =
  (stage2 opL' (codeLift2 opL') (stageB x1) (stageB x2))
stageB eL = (Now eL)

opCl' : {A : Set} ->
     ((ClMonoidTerm A) -> ((ClMonoidTerm A) -> (ClMonoidTerm A)))
opCl' x1 x2 = (opCl x1 x2)
eCl' : {A : Set} -> (ClMonoidTerm A)
eCl'  = eCl
stageCl : {A : Set} -> ((ClMonoidTerm A) -> (Staged (ClMonoidTerm A)))
stageCl (sing x1) = (Now (sing x1))
stageCl (opCl x1 x2) =
  (stage2 opCl' (codeLift2 opCl') (stageCl x1) (stageCl x2))
stageCl eCl = (Now eCl)
```

```
opOL' : {n : Nat} ->
     ((OpMonoidTerm n) -> ((OpMonoidTerm n) -> (OpMonoidTerm n)))
opOL' x1 x2 = (opOL x1 x2)
eOL' : {n : Nat} -> (OpMonoidTerm n)
eOL'  = eOL
stageOpB : {n : Nat} -> ((OpMonoidTerm n) -> (Staged (OpMonoidTerm n)))
stageOpB (v x1) = (const (code (v x1)))
stageOpB (opOL x1 x2) =
  (stage2 opOL' (codeLift2 opOL') (stageOpB x1) (stageOpB x2))
stageOpB eOL = (Now eOL)

opOL2' : {n : Nat} {A : Set} ->
    ((OpMonoidTerm2 n A) -> ((OpMonoidTerm2 n A) -> (OpMonoidTerm2 n A)))
opOL2' x1 x2 = (opOL2 x1 x2)
eOL2' : {n : Nat} {A : Set} -> (OpMonoidTerm2 n A)
eOL2'  = eOL2
stageOp : {n : Nat} {A : Set} ->
     ((OpMonoidTerm2 n A) -> (Staged (OpMonoidTerm2 n A)))
stageOp (sing2 x1) = (Now (sing2 x1))
stageOp (v2 x1) = (const (code (v2 x1)))
stageOp (opOL2 x1 x2) =
  (stage2 opOL2' (codeLift2 opOL2') (stageOp x1) (stageOp x2))
stageOp eOL2 = (Now eOL2)
```

```
record StagedRepr (A : Set) (Repr : (Set -> Set)) : Set where
  constructor repr
  field
    opT : ((Repr A) -> ((Repr A) -> (Repr A)))
    eT : (Repr A)
```

Bibliography

Abel, A., Allais, G., Hameer, A., Pientka, B., Momigliano, A., Schäfer, S., and Stark, K. (2019). Poplmark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming*, **29**.

Agda Library (2020). Agda Standard Library. https://github.com/agda/agda-stdlib. Version 1.4.

Al-hassy, M. (2019). Making Modules with Meta-Programmed Meta-Primitives. https://alhassy.github.io/next-700-module-systems/prototype/package-former.html#hundreds-of-theories. Accessed: 2019-11-20.

Al-hassy, M., Carette, J., and Kahl, W. (2019). A Language Feature to Unbundle Data at Will (Short Paper). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2019, pages 14 – 19, New York, NY, USA. ACM.

Algehed, M., Bernardy, J.-P., and Hritcu, C. (2020). Dynamic IFC theorems for free! arXiv:2005.04722. https://arxiv.org/abs/2005.04722.

Anand, A., Boulier, S., Cohen, C., Sozeau, M., and Tabareau, N. (2018). Towards

Certified Meta-Programming with Typed Template-Coq. In *ITP 2018 - 9th Conference on Interactive Theorem Proving*, volume 10895 of *LNCS*, pages 20–39, Oxford, United Kingdom. Springer.

Aspinall, D. and Hofmann, M. (2005). Dependent types. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 45–86. MIT press.

Autexier, S., Hutter, D., Mantel, H., and Schairer, A. (2000). Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, pages 73–88, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ballarin, C. (2003). Locales and Locale Expressions in Isabelle/Isar. In *International Workshop on Types for Proofs and Programs*, pages 34–50. Springer.

Ballarin, C. (2006). Interpretation of Locales in Isabelle: Managing Dependencies between Locales. Technical report, Technische Universität München (TUM).

Bauer, A. (2020). What makes dependent type theory more suitable than set theory for proof assistants? posted at: https://mathoverflow.net/questions/376839/what-makes-dependent-type-theory-more-suitable-than-set-theory-for-proof-assista.

Bercic, K., Carette, J., Farmer, W. M., Kohlhase, M., Müller, D., Rabe, F., and Sharoda, Y. (2020). The Space of Mathematical Software Systems – A Survey of Paradigmatic Systems. arXiv:2002.04955.

Betzendahl, J. and Kohlhase, M. (2018). Translating the IMPS Theory Library to

MMT/OMDoc. In F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, editors, *Intelligent Computer Mathematics*, pages 7–22, Cham. Springer International Publishing.

Bidoit, M. and Mosses, P. D. (2003). *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*, volume 2900. Springer.

Blanchette, J. C., Kaliszyk, C., Paulson, L. C., and Urban, J. (2016). Hammering towards QED. *Journal of Formalized Reasoning*, **9**(1), 101–148.

Blöndal, B., Löh, A., and Scott, R. (2018). Deriving Via: Or, How to Turn Hand-Written Instances into an Anti-Pattern. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 55–67, New York, NY, USA. Association for Computing Machinery.

Boyer, R. *et al.* (1994). The QED manifesto. *Automated Deduction–CADE*, **12**, 238–251.

Bracha, G. (1992). *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, The University of Utah.

Burstall, R. M. and Goguen, J. A. (1980). The Semantics of Clear, a Specification Language. In D. Bjøorner, editor, *Abstract Software Specifications*, pages 292 – 332, Berlin, Heidelberg. Springer Berlin Heidelberg.

Capretta, V. (1999). Universal Algebra in Type Theory. In *Theorem Proving in Higher Order Logics*, pages 131 – 148, Berlin, Heidelberg. Springer Berlin Heidelberg.

Carette, J. and Kiselyov, O. (2005). Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. In R. Glück and

M. Lowry, editors, *Generative Programming and Component Engineering*, pages 256–274, Berlin, Heidelberg. Springer Berlin Heidelberg.

Carette, J. and O'Connor, R. (2011a). MathScheme Library Declarations. GitHub repository https://github.com/JacquesCarette/MathScheme/blob/7f24a911790d67f5ab28db425bda1200bc0d5a45/prototype/src/Algebra/Base.msl.

Carette, J. and O'Connor, R. (2011b). Prototype of MathScheme Combinators. GitHub repository https://github.com/JacquesCarette/MathScheme/tree/7f24a911790d67f5ab28db425bda1200bc0d5a45/prototype.

Carette, J. and O'Connor, R. (2012). Theory Presentation Combinators. In *Intelligent Computer Mathematics*, volume 7362 of *Lecture Notes in Computer Science*, pages 202–215. Springer Berlin Heidelberg.

Carette, J., Kiselyov, O., and Shan, C.-c. (2009). Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming*, **19**(5), 509 – 543.

Carette, J., Elsheikh, M., and Smith, S. (2011a). A Generative Geometric Kernel. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 53 – 62. ACM.

Carette, J., Farmer, W. M., Jeremic, F., Maccio, V., O'Connor, R., and Tran, Q. (2011b). The MathScheme Library: Some Preliminary Experiments. *arXiv:1106.1862*.

Carette, J., Farmer, W. M., and Sharoda, Y. (2018). Biform Theories: Project Description. In F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, editors, *Intelligent Computer Mathematics*, pages 76–86, Cham. Springer International Publishing.

Carette, J., O'Connor, R., and Sharoda, Y. (2019). Building on the Diamonds between Theories: Theory Presentation Combinators. arXiv:1812.08079. Submitted to Journal of Automated Reasoning.

Carette, J., Farmer, W. M., Kohlhase, M., and Rabe, F. (2020a). Big Math and the One-Brain Barrier – The Tetrapod Model of Mathematical Knowledge. *Mathematical Intelligencer*.

Carette, J., Farmer, W. M., and Sharoda, Y. (2020b). Leveraging the Information Contained in Theory Presentations. In C. Benzmüller and B. Miller, editors, *Intelligent Computer Mathematics*, pages 55–70, Cham. Springer International Publishing.

Cartmell, J. (1986). Generalised Algebraic Theories and Contextual Categories. *Annals of Pure and Applied Logic*, **32**, 209 – 243.

Christiansen, D. and Brady, E. (2016). Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 284–297, New York, NY, USA. Association for Computing Machinery.

Cohen, C., Sakaguchi, K., and Tassi, E. (2020). Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi (System Description). In Z. M. Ariola, editor,

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:21, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Cornes, C. and Terrasse, D. (1996). Automating Inversion of Inductive Predicates in Coq. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, pages 85–104, Berlin, Heidelberg. Springer Berlin Heidelberg.

Crary, K. (2005). Logical relations and a case study in equivalence checking. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 6, pages 223–244. MIT Press, Cambridge, MA.

Cruz-Filipe, L. and Spitters, B. (2003). Program Extraction from Large Proof Developments. In *International Conference on Theorem Proving in Higher Order Logics*, pages 205–220. Springer.

Czajka, Ł. and Kaliszyk, C. (2018). Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, **61**(1-4), 423–453.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison Wesley.

Czarnecki, K., Helsen, S., and Eisenecker, U. (2005). Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, **10**(2), 143–169.

Dowek, G. and Thiré, F. (2019). Logipedia: a multi-system encyclopedia of formal proofs. http://www.lsv.fr/~dowek/Publi/logipedia.pdf.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A Mechanism for Fine-grained Reuse. *ACM Trans. Program. Lang. Syst.*, **28**(2), 331–388.

Dunchev, C., Guidi, F., Sacerdoti Coen, C., and Tassi, E. (2015). ELPI: Fast, Embeddable, λProlog Interpreter. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 460–468, Berlin, Heidelberg. Springer Berlin Heidelberg.

Durán, F. and Meseguer, J. (2007). Maude's Module Algebra. *Science of Computer Programming*, **66**(2), 125–153.

Ebner, G., Ullrich, S., Roesch, J., Avigad, J., and de Moura, L. (2017). A Metaprogramming Framework for Formal Verification. *Proceedings of the ACM on Programming Languages*, **1**(ICFP), 1–29.

Ede, D. (2019). Allow metaprogramming to generate top level definitions other than functions. Github issue 3699: https://github.com/agda/agda/issues/3699. Accessed: December 2,2020.

Ehrig, H. and Mahr, B. (1985). *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg.

Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. Academic Press.

Farmer, W. M. (1994). Theory Interpretation in Simple Type Theory. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting*, pages 96–123, Berlin, Heidelberg. Springer Berlin Heidelberg.

Farmer, W. M. (2004). MKM: A New Interdisciplinary Field of Research. *ACM SIGSAM Bulletin*, **38**(2), 47 – 52.

Farmer, W. M. (2013). The Formalization of Syntax-Based Mathematical Algorithms Using Quotation and Evaluation. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 35–50, Berlin, Heidelberg. Springer Berlin Heidelberg.

Farmer, W. M., Guttman, J. D., and Thayer, F. J. (1992). Little Theories. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 567 – 581, London, UK. Springer-Verlag.

Farmer, W. M., Guttman, J. D., and Thayer, F. J. (1993). IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, **11**(2), 213–248.

Ganz, S. E., Sabry, A., and Taha, W. (2001). Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. *ACM SIGPLAN Notices*, **36**(10), 74–85.

Garillot, F., Gonthier, G., Mahboubi, A., and Rideau, L. (2009). Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics*, pages 327 – 342, Berlin, Heidelberg. Springer Berlin Heidelberg.

Geuvers, H., Pollack, R., Wiedijk, F., and Zwanenburg, J. (2002). A Constructive Algebraic Hierarchy in Coq. *Journal of Symbolic Computation*, **34**(4), 271 – 286.

Goguen, J. A., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J.-P. (2000). Introducing obj. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Springer US, Boston, MA.

Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Théry, L. (2013). A machine-checked proof of the odd order theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 163–179, Berlin, Heidelberg. Springer Berlin Heidelberg.

Grabowski, A., Korniłowicz, A., and Schwarzweller, C. (2015). Equality in Computer Proof-Assistants. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 45–54.

Gries, D. and Schneider, F. B. (1993). Propositional Calculus. In *A Logical Approach to Discrete Math*, pages 41–61. Springer New York, New York, NY.

Gross, J., Kubota, K., Mechveliani, S. D., *et al.* (2020). Why dependent type theory? https://coq.discourse.group/t/why-dependent-type-theory/657.

Gunther, E., Gadea, A., and Pagano, M. (2018). Formalization of Universal Algebra in Agda. *Electronic Notes in Theoretical Computer Science*, **338**, 147 – 166. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

Haftmann, F. and Nipkow, T. (2010). Code Generation via Higher-Order Rewrite Systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, pages 103–117, Berlin, Heidelberg. Springer Berlin Heidelberg.

Haskell Wiki (2015). Functor-Applicative-Monad Proposal. [Online; accessed 14-November-2019].

Iancu, M., Kohlhase, M., Rabe, F., and Urban, J. (2013). The Mizar Mathematical Library in OMDoc: Translation and Applications. *Journal of Automated Reasoning*, **50**(2), 191–202.

Jacobson, N. (1985). *Basic Algebra I*. W. H. Freeman and Company.

Jipsen, P. (2019). List of Mathematical Structures. http://math.chapman.edu/~jipsen/structures/doku.php. Accessed: March 20, 2020.

Kaliszyk, C. and Pąk, K. (2018). Isabelle Import Infrastructure for the Mizar Mathematical Library. In F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, editors, *Intelligent Computer Mathematics*, pages 131–146, Cham. Springer International Publishing.

Kaliszyk, C. and Pak, K. (2019). Declarative Proof Translation (Short Paper). In J. Harrison, J. O'Leary, and A. Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:7, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Kohlhase, M. and Rabe, F. (2016). QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge. *Journal of Formalized Reasoning*, **9**(1), 201–234.

Kohlhase, M., Rabe, F., and Zholudev, V. (2010). Towards MKM in the Large: Modular Representationand Scalable Software Architecture. In *International Conference on Intelligent Computer Mathematics*, pages 370–384. Springer.

Lämmel, R. and Jones, S. P. (2003). Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. *ACM SIGPLAN Notices*, **38**(3), 26–37.

Lens Library (2020). Haskell Lens Library. https://hackage.haskell.org/package/lens. version 4.19.1; Accessed: 2020-03-22.

Letouzey, P. (2003). A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, pages 200–219, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lewis, R. Y. (2017). An Extensible Ad Hoc Interface between Lean and Mathematica. *Electronic Proceedings in Theoretical Computer Science*, **262**, 23–37.

Liesnikov, B., Ullrich, M., and Forster, Y. (2020). Generating induction principles and subterm relations for inductive types using MetaCoq. arXiv:2006.15135.

Lilis, Y. and Savidis, A. (2019). A survey of metaprogramming languages. *ACM Comput. Surv.*, **52**(6).

Magalhães, J. P., Dijkstra, A., Jeuring, J., and Löh, A. (2010). A Generic Deriving Mechanism for Haskell. *ACM SIGPLAN Notices*, **45**(11), 37–48.

Mahboubi, A. and Tassi, E. (2013). Canonical Structures for the Working Coq User. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 19–34, Berlin, Heidelberg. Springer Berlin Heidelberg.

Mahboubi, A. and Tassi, E. (2020). *Mathematical Components*. Zenodo.

Mazur, B. (2008). When is One Thing Equal to Some Other Thing? *Proof and other dilemmas: Mathematics and philosophy*, **59**, 221.

Mazzoli, F. and Abel, A. (2016). Type Checking through Unification. arXiv:1609.09709.

Mazzoli, F., Danielsson, N. A., Norell, U., Vezzosi, A., and Abel, A. (2017). Tog, a prototypical implementation of dependent types. GitHub Repository https://github.com/bitonic/tog.

McKenzie, R. N., McNulty, G. F., and Taylor, W. F. (1987). *Algebras, Lattices, Varieties*, volume 1. American Mathematical Soc.

Meinke, K. and Tucker, J. V. (1993). Universal Algebra. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 1): Background: Mathematical Structures*, page 189–368. Oxford University Press, Inc., USA.

Mosses, P. D. (2004). *CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg.

Mulligan, D. P., Owens, S., Gray, K. E., Ridge, T., and Sewell, P. (2014). Lem: Reusable Engineering of Real-world Semantics. *ACM SIGPLAN Notices*, **49**(9), 175–188.

Müller, D., Rothgang, C., Liu, Y., and Rabe, F. (2017). Alignment-based Translations Across Formal Systems Using Interface Theories. *Electronic Proceedings in Theoretical Computer Science*, **262**, 77–93.

nLab authors (2020a). colimit. http://ncatlab.org/nlab/show/colimit. Revision 17.

nLab authors (2020b). pushout. http://ncatlab.org/nlab/show/pushout. Revision 22.

Pierce, B. C. (1990). *A Taste of Category Theory for Computer Scientists*. Carnegie Mellon University.

Pitts, A. M. (2001). Categorical Logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 5): Logic and algebraic methods*, page 39–123. Oxford University Press, Inc., USA.

Plotkin, G. and Abadi, M. (1993). A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, pages 361–375, Berlin, Heidelberg. Springer Berlin Heidelberg.

Pollack, R. (2002). Dependently Typed Records in Type Theory. *Formal Aspects of Computing*, **13**(3-5), 386–402.

Pottier, L. (2019). Coq User Contributions - Algebra Library. GitHub Repository https://github.com/coq-contribs/algebra v8.10.0.

Rabe, F. and Kohlhase, M. (2013a). A Scalable Module System. *Information and Computation*, **230**, 1–54.

Rabe, F. and Kohlhase, M. (2013b). A scalable module system. *Inf. Comput.*, **230**, 1–54.

Rabe, F. and Sharoda, Y. (2019). Diagram Combinators in MMT. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 211–226, Cham. Springer International Publishing.

Rebours, N. (2019). An Introduction to OCaml PPX Ecosystem. Tutorial: https://tarides.com/blog/2019-05-09-an-introduction-to-ocaml-ppx-ecosystem. Accessed on December 2020.

Reynolds, J. C. (1983). Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, pages 513–523.

Sakaguchi, K. (2020). Validating Mathematical Structures. arXiv:2002.00620.

Sakkinen, M. (1989). Disciplined Inheritance. In *ECOOP*, volume 89, pages 39–56.

Sannella, D. and Tarlecki, A. (2012). Category theory. In *Foundations of Algebraic Specification and Formal Software Development*, Monographs in Theoretical Computer Science. An EATCS Series, pages 97 – 153. Springer Berlin Heidelberg.

Scrap Your Boilerplate (2019). Haskell scrap-your-boilerplate package. https://hackage.haskell.org/package/syb. version 0.7.1; Accessed: 2020-11-18.

Sharoda, Y. (2019). Leveraging Information Contained in Theory Presentations. In *Workshop Papers at 12th Conference on Intelligent Computer Mathematics CICM 2019*, volume 2634. CEUR Workshop Proceedings. http://ceur-ws.org/Vol-2634/DP7.pdf.

Sheard, T. (2001). Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, pages 2–44, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sheard, T. and Jones, S. P. (2002). Template Meta-Programming for Haskell. In

Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell '02, page 1–16, New York, NY, USA. Association for Computing Machinery.

Sheard, T., Benaissa, Z.-e.-a., and Pasalic, E. (2000). DSL Implementation Using Staging and Monads. *ACM SIGPLAN Notices*, **35**(1), 81–94.

Shulman, M. (2010). In Praise of Dependent Types. The n-Category Café blog. https://golem.ph.utexas.edu/category/2010/03/in_praise_of_dependent_types.html.

Smith, D. R. (1999). Mechanizing the Development of Software. In M. Broy and R. Steinbrueggen, editors, *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, pages 251–292. IOS Press, Amsterdam.

Spitters, B. and van der Weegen, E. (2010). Developing the Algebraic Hierarchy with Type Classes in Coq. In *Interactive Theorem Proving*, pages 490 – 493, Berlin, Heidelberg. Springer Berlin Heidelberg.

Spitters, B. and Van der Weegen, E. (2011). Type Classes for Mathematics in Type Theory. *arXiv preprint arXiv:1102.1323*.

Taha, W. (1999). *Multi-Stage Programming: Its Theory and Applications*. Ph.D. thesis, Oregon Graduate Institute of Science and Technology.

Tarski, A., Mostowski, A., and Robinson, R. M. (1953). *Undecidable Theories*, volume 13. Elsevier.

Tassi, E. (2018). Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). In *Fourth International Workshop on Coq for Programming Languages*.

Tassi, E. (2019). Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. In *ITP 2019 - 10th International Conference on Interactive Theorem Proving*, Portland, United States.

Team, T. M. (2019). The Lean Mathematical Library. *arXiv: 1910.09336*. https://arxiv.org/abs/1910.09336.

van der Walt, P. (2012). *Reflection in Agda*. Master's thesis, Utrecht University.

Wadler, P. (1989). Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359.

Wadler, P. (2003). A Prettier Printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243.

Whitehead, A. (1898). *A Treatise on Universal Algebra: with Applications*. Cornell University Library historical math monographs. The University Press.

Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D., Paige, R., Lauder, M., Schürr, A., and Wagelaar, D. (2011). A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. In J. Cabot and E. Visser, editors, *Theory and Practice of Model Transformations*, pages 31–46, Berlin, Heidelberg. Springer Berlin Heidelberg.

Yallop, J. (2016). Staging Generic Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, page 85–96, New York, NY, USA. Association for Computing Machinery.