# Do-it-yourself Module Systems

## Extending Dependently-Typed Languages to Implement Module System Features In The Core Language

Department of Computing and Software

McMaster University

Musa Al-hassy

October 14, 2020

PhD Thesis

```
-- Supervisors                                    -- Emails
Jacques Carette                                   carette@mcmaster.ca
Wolfram Kahl                                      kahl@cas.mcmaster.ca
```

## Abstract

Can parameterised records and algebraic datatypes —i.e., $\Pi$-, $\Sigma$-, and $\mathcal{W}$-types— be derived from one pragmatic declaration?

Record types give a universe of discourse, parameterised record types fix parts of that universe ahead of time, and algebraic datatypes give us first-class syntax, whence evaluators and optimisers.

The answer is in the affirmative. Besides a practical shared declaration interface, which is extensible in the language, we also find that common data structures correspond to simple theories.

# Contents

# List of Tables

# 1 Introduction

The construction of programming libraries is managed by decomposing ideas into self-contained units called 'packages' whose relationships are then formalised as transformations that reorganise representations of data. Depending on the *expressivity* of a language, packages may serve to avoid having different ideas share the same name —which is usually their *only* use— but they may additionally serve as silos of source definitions from which interfaces and types may be *extracted*. Figure **??** exemplifies the idea for monoids —which themselves model a notion of composition. In general, such derived constructions are *out of reach* from *within* a language and have to be extracted *by hand* by users who have the time and training to do so. Unfortunately, this is the standard approach; even though it is error-prone and disguises mechanical *library methods* (that are written *once* and proven correct) as *design patterns* (which need to be carefully implemented for *each* use and argued to be correct). The goal of this thesis is to show that sufficiently expressive languages make packages an interesting *and* central programming concept by extending their common use as silos of data with the ability for *users* to *mechanically* derive related ideas (programming constructs) as well as the relationships between them.

The framework developed in this thesis is motivated by the following concerns when developing libraries in the dependently-typed language (DTL) Agda, such as [**RATH**].

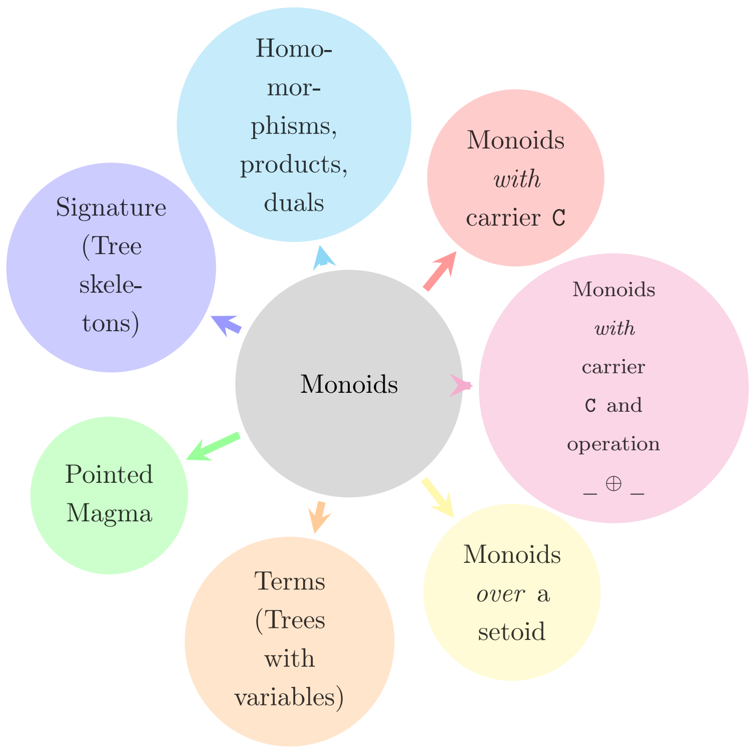
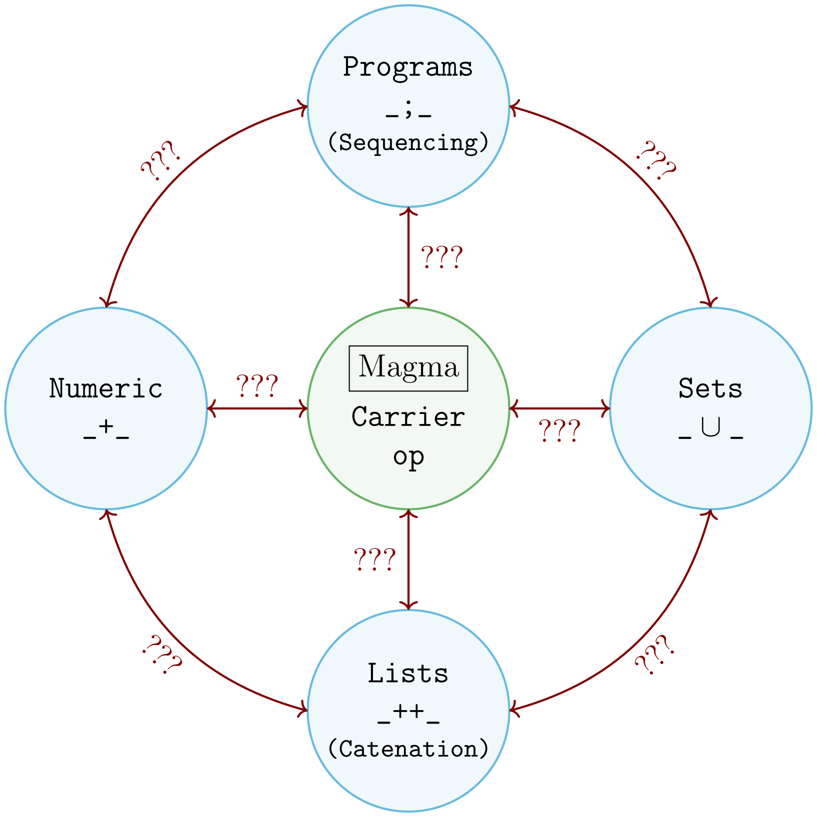


Figure 1.1: Deriving related *types* from *the* definition of monoids

## 1.1 Practical Concern ♯1: Renaming and Remembering Relationships

There is excessive repetition in the simplest of tasks when working with packages; e.g., to *uniformly* decorate the names in a package with subscripts $_0$, $_1$, $_2$ requires the package's contents be listed thrice. It would be more economical to *apply* a renaming *function* to a package. Even worse, as show in Figure **??**, sometimes we want to perform a renaming to view an idea in a more natural, concrete, setting; but shallow renaming mechanisms *lose the relationships* to the original parent package and so 'do nothing' coercions have to be written by hand.

Figure 1.2: Given green, derive cyan candidate constructions, require red relationships



The need to 'remember relationships' is shared by the other concerns discussed in this section.

## 1.2 Practical Concern ♯2: Unbundling

In general, in a DTL, *packages behave like functions* in that they may have a subset of their contents designated as *parameters exposed at the type-level* which users can *instantiate*. The shift between the two forms is known as **the unbundling problem packaging_mathematical_structu**
Unfortunately, library developers generally provide only a few *variations* on *a* package; such as having no parameters or having only *functional symbols* as parameters —c.f., the carrier `C` and operation `_⊕_` in Figure ??. Whereas functions can *bundle-up* or *unbundle* their parameters using currying and uncurrying, only the latter is generally supported and, even then, not in an elegant fashion. Rather than provide *several variations* on a package, it would be more economical to provide one singular fully-bundled package and have an operator that allows users to *declaratively*, "on the fly", expose package constituents as parameters.

Let us clarify this subtlety.

At its core, the unbundling problem is well-known as '(un)currying': The restructuring of record consuming functions as 'parameterised families of functions' as follows.

$$\frac{I : \mathsf{Type} \qquad X : \mathsf{Type} \qquad Y : \mathsf{Type}}{I \times X \to Y \quad \cong \quad I \to (X \to Y)}$$

The right side brings a number of *practical conveniences* in the form of simplified concrete syntax —e.g., reduced parenthesise for function arguments— and in terms of auxiliary combinators to 'fix' an $I$-value ahead of time —i.e., 'partial function application'. The unbundling problem replaces simple product and function types with their *dependent* generalisations (to be defined and discussed in Chapter 2, the background):

$$\frac{I : \mathsf{Type} \qquad X : I \to \mathsf{Type} \qquad Y : (\Sigma i : I \bullet X\, i) \to \mathsf{Type}}{\Pi p : (\Sigma i : I \bullet X\, i) \bullet Y\, p \quad \cong \quad \Pi i : I \bullet \Pi x : X\, i \bullet Y\, (i, x)}$$

As with currying, the right side here is preferable at times since it immediately lets one 'fix' —i.e., select— a value $i_0 : I$ to obtain the specialised type

$$\Pi x : X\, i_0 \bullet Y\, (i_0, x)$$

In contrast, the left side can only be controted to simulate the idea of fixing a field ahead of time; e.g.:

$$\Pi p : (\Sigma i : I \bullet X\, i) \bullet Z\, p \quad \text{where} \quad Z\, p \;=\; Y\, p \times (\mathsf{fst}\, p \;\equiv\; i_0)$$

Figure 1.3: Bundled forms: Two solid arrows to get one dashed arrow

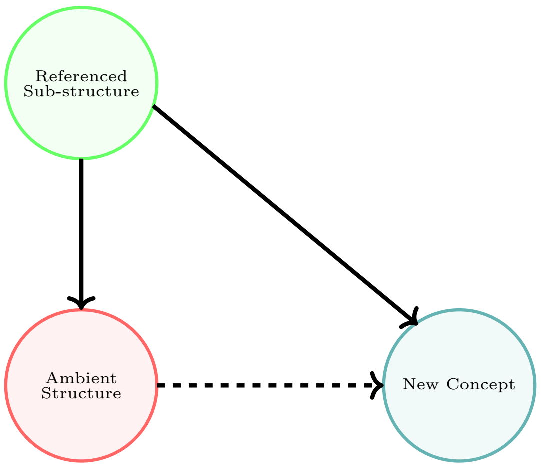
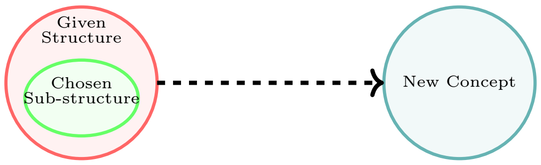


Figure 1.4: Unbundled forms: Obtain the dashed arrow explicitly



The verbosity of this formulation is what we wish to mitigate.

The dependent nature of DTLs means that this problem is not solely about functions —and so, we cannot simply insist on formulations similar to the right side; i.e., omitting the record former '$\Sigma$'. Since types can *depend on the values* of other types, this now becomes a problem about types as well. In particular, we may view the parameterised type family $Z$ as being a new concept that is formed around a chosen substructure $i_0 : X$ —which must be referenced from 'outside' using the ambient structure $Y$. It would be far more practical to treat the structure we actually care about as if it were a 'top level item' rather than 'something to be hunted down'. These situation is captured nicely by informal diagrams[1] of Figures 1.3 and 1.4; concrete instances of these diagrams are presented later on, as Figures **??** and 3.1.

It is interesting to note that the unbundling problem appears in a number of guises within the setting of programming language design. We briefly mention two scenarios.

1. "De-structuring" or "Pattern Matching": Instead of writing `f p =` $\cdots$`fst p`$\cdots$ to define a function `f` working on pairs `p` that makes a reference to the first field, we may instead

[1]In these diagrams, the arrows are used to denote a dependency relationship.

write `f (x, y) = ···x···` and access the first field directly. This pattern is popular in modern languages, such as Haskell and JavaScript.

The shift from `p` being **explicitly** a *packaging mechanism* to to it **silently** disappearing in the *presentation* of the function definition is one of the pragmatic aims that we strive for with our system.

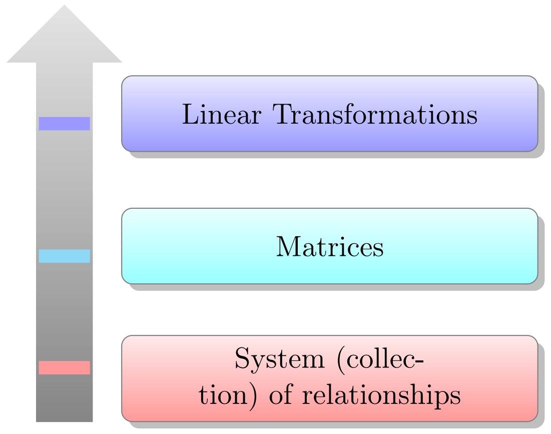
2. "Quantifier Nesting": Instead of writing $\lambda$ `p` $\bullet$ ··· `fst p` ··· `snd p` ··· for a functional expression operating on a pair, some languages permit writing $\lambda$ `x` $\bullet$ $\lambda$ `y` $\bullet$ ··· `x` ··· `y` ···. The problem is that many languages treat these differently and require a combinator to shift between the two; the former being more conventional in functional settings whereas the latter is more common in imperative and object-oriented settings.

   For $\lambda$, the idea is to curry; but when one speaks about types and so replaces the $\lambda$ with $\forall$ —i.e., $\Pi$— then things are not as clear cut and there is no accepted combinator to switch between the two forms. This is what our system aims to rectify.

## 1.3 Theoretical Concern ♯1: Exceptionality

DTLs blur the distinction between expressions and types, treating them as the same thing: *Terms*. This collapses a number of seemingly different language constructs into the same thing —e.g., programs and proofs are essentially the same thing. Unfortunately, packages are treated as *exceptional* values that differ from *usual* values —such as functions and numbers— in that the former are 'second-class citizens' which only serve to collect the latter 'first-class citizens'. This forces users to learn two families of 'sub-languages' —one for each citizen class. There is essentially no *theoretical* reason why packages do not deserve first-class citizenship, and so receive the same treatment as other *unexceptional* values. Another advantage of giving packages equal treatment is that we are inexorably led to wonder what **computable algebraic structure** they have and how they relate to other constructs in a language; e.g., packages are essentially record-valued functions.

Perhaps the most famous instance of how the promotion of a second-class concept to first-class status comes from linear algebra, and subsequently, the theory of vector spaces. When there are a number of relationships involving a number of unknowns, the relationships could be 'massaged algebraically' to produce simper constraints on the unknowns, possibly providing 'solutions' to the system of relationships directly. The shift from *systems of equations* that serve to collect relationships, to *matrices* gave way to the treatment of such systems as algebraic entities unto themselves: They can be treated with nearly the same interface as that of integers, say, that of rings. As such, 'component-wise addition of equations in system $A$ with system $B$" becomes more tractable as $A + B$ and satisfies the many familiar properties of numeric addition. Even more generally, for any theory of 'individuals' one can consider the associated matrix theory —e.g., if $M$ is a monoid, then the matrices whose elements are drawn from $M$ *inherits* the monoidal structure— and so gives a construction of *system of* *equations* on that theory. To investigate the algebraic nature of packaging mechanisms is another aim of this thesis.



An interesting aside is that a *collection* mechanism gave rise to the abstract *matrix* concept, which is then seen as a reification of the even more abstract notion of linear transformation between vector spaces —-which are in turn, packages paramterised over fields (and, in practice, over basis).

## 1.4 Theoretical Concern ♯2: Syntax

It is well known that sequences of declarations may be grouped together within a *package*. If any declarations are opaque, not fully undefined, they become *parameters* of the package —which may then be identified as a *record type* with the opaque declarations called *fields*. However, when a declaration is *intentionally opaque* not because it is missing an implementation, but rather it acts as a value construction itself then one uses *algebraic data types*, or 'termtypes'. Such types share the general structure of a package, as shown in the codeblock below, and so it would be interesting to illuminate the exact difference between the concepts —*if any*. In practice, one forms a record type to model an interface, instances of which are actual implementations, and forms an *associated* termtype to *describe computations* over that record type, thereby making available a syntactic treatment of the interface —textual substitution, simplification / optimisation, evaluators, canonical forms. For example, as shown in figure **??**, the record type of monoids models composition whereas the (tremendously useful) termtype of binary trees acts as a description language for monoids. The *problem of maintenance* now arises: Whenever the record type is altered, one must mechanically update the associated termtype. It would be more economical to extract *both* record types and termtypes from a single package declaration.

Spot the difference

Theory of monoids

```
record Monoid : Set₁ where
    C : Set
    -- function symbols
    _;_ : C → C → C
    Id : C
    -- axioms
    lid : ∀ x → Id ; x ≡ x
    rid : ∀ x → x ; Id ≡ x
    assoc : ∀ x y z
          →   (x ; y) ; z
            ≡ x ; (y ; z)
```

Terms over 'variables' `C`

```
data Term (C : Set) : Set where
    -- injection
    embed : C → Term C
    -- function symbols
    _;_  : Term C → Term C → Term
    ↪  C
    Id : Term C
```

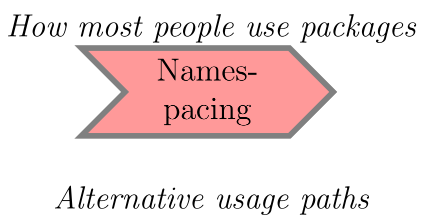
Binary trees with leaf labels drawn from `C`

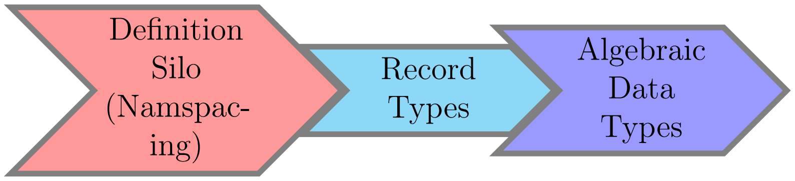
```
data Trees (C : Set) : Set where
    Nil    : Tree C
    Leaf   : C → Tree C
    Branch : Tree C
           → Tree C → Tree C
```

## 1.5 Guiding Principle: Practical Usability

In this thesis, we aim to mitigate the above concerns with a focus on **practicality**. A theoretical framework may address the concerns, but it would be incapable of accommodating *real-world use-cases* when it cannot be applied to real-world code. For instance, one may speak of 'amalgamating packages', which can always "be made disjoint", but in practice the union of two packages would likely result in name clashes which could be avoided in a number of ways but the *user-defined names* are important and so a result that is "unique up to isomorphism" is not practical. As such, we will implement a framework to show that the above concerns can be addressed in a way that **actually works**. A concrete example is demonstrated later on, such as in Figure ??.

## 1.6 Thesis Overview





The remainder of the thesis is organised as follows.

◇ Chapter 2 consists of preliminaries, to make the thesis self-contained, and contributions of the thesis.

  A review of dependently-typed programming with Agda is presented, with a focus on its packaging constructs: Namespacing with `module`, record types with `record`, and as contexts with Σ-padding. The interdefinability of the aforementioned three packaging constructs is demonstrated. After-which is a quick review of other DTLs that shows the idea of a unified notion of package is promising —Agda is only a presentation language, but the ideas transfer to other DTLs.

  With sufficient preliminaries reviewed, the reader is in a position to appreciate a survey of package systems in DTLs and the contributions of this thesis. The contributions listed will then act as a guide for the remainder of the thesis.

◇ Chapter 3 consists of real world examples of problems encountered with the existing package system of Agda.

  Along the way, we identify a set of *DTL design patterns* that users repeatedly implement. An indicator of the **practicality** of our resulting framework is the ability to actually implement such patterns as library methods.

◇ Chapter 4 discusses a prototype that addresses *nearly* all of our concerns.

  Unfortunately, the prototype introduces a new sublanguage for users to learn. Packages are *nearly* first-class citizens: Their manipulation must be specified in Lisp rather than in the host language, Agda. However, the ability to rapidly, textually, manipulate a package makes the prototype an extremely useful tool to test ideas and implementations of package combinators. In particular, the aforementioned example of forming unions of packages is implemented in such a way that the amount of input required —such as *along* what interface should a given pair of packages be *glued* and *how* name clashes should be handled— can be 'inferred' when not provided by making use of Lisp's support for keyword arguments. Moreover, the union operation is a *user-defined* combinator: It is a *possible* implementation by a user of the prototype, built upon the prototype's "package meta-primitives".

◇ Chapter 5 takes the lessons learned from the prototype to show that *DTLs can have a unified package system within the host language.*

  The prototype is given semantics as Agda types and functions by forming a **practical** library within Agda that achieves the core features of the prototype. The switch to

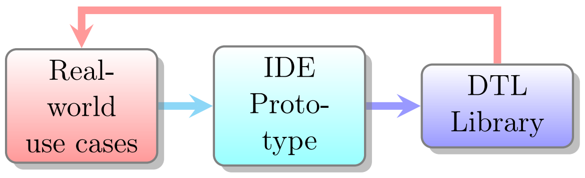


Figure 1.5: Approach for a **practical** framework

a DTL is nontrivial due to the type system; e.g., fresh names cannot be arbitrarily introduced nor can syntactic shuffling happen without a bit of overhead. The resulting library is both usable and practical, but lacks the immense power of the prototype due to the limitations of the existing implementation of Agda's metaprogramming facility.

We conclude with the observation that ubiquitous data structures in computing arise *mechanically* as termtypes of simple 'mathematical theories' —i.e., packages.

◇ Chapter 6 concludes with a discussion about the results presented in the thesis.

The underlying motivation for the research is the conviction that packages play *the* crucial role for forming compound computations, subsuming *both* record types and termtypes. The approach followed is summarised in Figure **??**.

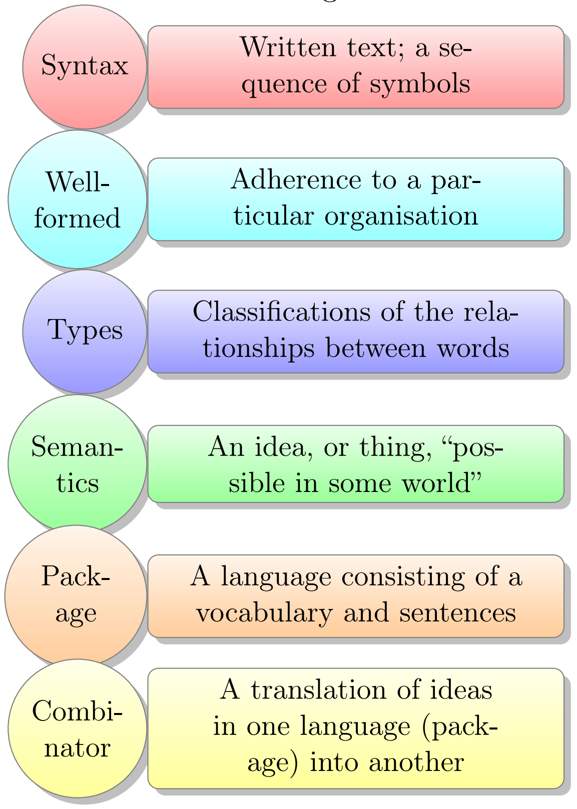
How accessible is this thesis?

- Chapter 1, this section, is presented from a high-level overview and tries to be accessible to a computer scientist exposed to fundamental functional programming.
- Chapter 2 tries to be **accessible to the layman**. It goes out of its way to explain basic ideas using analogies and 'real-life (non-computing) examples'. *The effort placed therein is so that 'almost anyone' can pick up this thesis and have 'an idea' of the problems it targets.*
- Chapter 3 may be tough reading for readers not familiar with Category Theory or having actually written any Agda code.
- Chapter 4 may be less daunting than Chapter 3, as it has line-by-line explanations of code fragments as well as accompanying diagrams.
- Chapter 5 tries to leave it to the reader on "how to read the chapter". The exposition of core ideas is presented in a box consisting of the main insight (operation definition) along with its realisation using Agda's metaprogramming mechanism. As such, readers could read the high level idea or the implementation —which, unlike Chapter 4, we have included so as to demonstrate that we are speaking of ideas whose implementations are not 'so difficult' that they apply to other DTLs besides Agda.
- Chapter 6, the final section, is a high-level overview of what has been accomplished and what we can look forward to achieving in the future. It may be slightly less accessible than Chapter 1.

# 2

# Packages and Their Parts

The purpose of language is to communicate ideas that 'live' in our minds —conversely, language also limits the kinds of thoughts we may have. In particular, written text captures ideas independently of the person who initially thought of them. To understand the idea *behind* a written sentence, people agree on **how** sentences may be organised and **what** content they denote from their parts. For example, in English, a sentence is considered 'well-formed' if it is in the order subject-verb-object —such as *"Jim ate the apple"*— and it is considered 'meaningful' if the subject and object are noun phrases that *denote things in a world that **could exist*** and the verb is a **possible action** by the subject on the object. For instance, in the previous example, there *could* be a person named *Jim* who *could* eat an apple, and so the sentence is meaningful. In contrast the phrase *"the colourless green apple kissed Jim"* is well-formed *but not* meaningful: The indicated action **could happen**, say, *in a world* of sentient apples; however, the subject *—the colourless green apple—* **cannot possibly exist** since a thing cannot be both lacking colour but also having colour at the same time. Moreover, *depending on who you ask*, the action of the previous example *—the [. . . ] apple **kissed** Jim—*, may be ludicrous *on the basis* that kissing is 'classified' as a verb whose subject, in the 'real' world, has the ability to kiss. As such, 'meaningfulness' is not necessarily fixed, but may vary. Likewise, as there is no one universal language spoken by all people, written text is also not fixed but varies; e.g., a translation tool may convert an idea *captured in* Arabic to a related idea *captured in* French. It is with these observations that we will discuss the concepts required to have a formal theory of packages, as summarised in the figure below.


Syntax
Written text; a sequence of symbols
Well-formed
Adherence to a particular organisation
Types
Classifications of the relationships between words
Semantics
An idea, or thing, "possible in some world"
Package
A language consisting of a vocabulary and sentences
Combinator
A translation of ideas in one language (package) into another

**Game-Play Analogy**

The contents of above figure may be intimidating to the uninitiated; so we reach for a game-play based analogy to further make the concepts accessible.

Programming, as is the case with all of mathematics, is the manipulation of symbols according to specific *rules*. Moreover, like a game, when one plays —i.e., shuffles symbols around— one may interpret the game pieces and the actions to *denote* some meaning, such as reflecting aspects of the players or of reality. Many play because it is fun to do so; there are only pieces (mathematical symbols or *terms*) and rules to be followed, and nothing more. Complex games may involve a number of pieces (terms) which are classified by the *types* of roles they serve, and the rules of play allow us to make observations or *judgements* about them; such as, "in the stage $\Gamma$ of the game, game piece $x$ serves the role $\tau$" and this is denoted $\Gamma \vdash x : \tau$ mathematically. Games which allow such observations are called *type theories* in mathematics. When games are played, they may override concepts in reality; e.g., in Chess, the phrase *Knight's move* refers to a particular set of possible plays and has nothing to do with knights in the real-world. As such, one calls the collection of specific game words, and what they mean, within a game (*type theory*) the *object-language* and uses the phrase *meta-language* to refer to the ambient language of the real-world. As it happens, some games have localised interactions between players where the rules may be changed temporarily and so we have *games within games*, then the object-language of the main game becomes the meta-language of the inner game. The rules of the game are its *syntax* and what the game means is its *semantics*. To say that a game piece (term) denotes some idea **I**, we need to be able to *express* that idea which may only be possible in the meta-language; e.g., pieces in a mini-game within a game may themselves denote pieces within the primary game —more concretely, a game may require a roll of a die whose numbers *denote*, or *refer to*, players in the main game which are not expressible in the mini-game. A *model* of a game (type theory) is an interpretation of the game's pieces in way that the rules are true under the interpretation.

To see an example of packages, consider the following real-world examples of dynamical systems. First, suppose you have a machine whose actions you cannot see, but you have a control panel before you that shows a starting screen, `start`, and the panel has one button, `next`, that forces the machine to act which updates the screen. Moreover, there is a screen capture called `thrice` *which happens* to be the result of pressing `next` three times after starting the machine. Second, suppose you are an artist mixing colours together.

Machine

```
State  : Type
start  : State
next   : State → State
thrice : State
thrice = next (next (next start))
```

Colours

```
Colour : Type
red    : Colour
green  : Colour
blue   : Colour
mix    : Colour × Colour → Colour
violet : Colour
violet = mix green blue
dark   : Colour → Colour
dark c = mix c blue
```

Each of these is a **package**: A sequence of ‘declarations’ of operations; wherein elements may be ‘parameters’ in the declarations of others. A **declaration** is a “name : classification” pair of words, *optionally* with another “name = definition” pair of words that shows how the new word *name* can be obtained from the vocabulary already declared thus far. For example, in these packages (languages) `thrice` and `violet` are aliases for expressions (sentences) constructed from other words. A **parameter** —also known as a **field**— is a declaration that is not an alias; i.e., it has no associated =-pair. Parameters are essentially the building blocks of a language; they cannot be expressed in terms of other words. A non-parameter is essentially *fully defined, implemented,* as an alias of a mixture of earlier words; whereas parameters are ‘opaque’ —*not yet implemented.* In particular, in the colours example above, `dark` *defines* a function that uses the *symbolic name* `mix` in its definition. There is an important subtlety between `mix` and `dark`: The latter, `dark`, is an *actual function* that is fully determined when an *implementation* of the *symbolic name* `mix` is provided. The (parameter) name `mix` is said to be a *function symbol* rather than a function: It is the *name* of a function, but it lacks any implementation and is thus not actually a function. A *function symbol* is to a function, like a name is to a person: Your name does not fully determine who you are as a person.

## Subsection Goals

This section aims to present a mathematical formalisation of packages. For brevity, we only consider parameters in the first few sections then accommodate non-parameters after a working definition is established. As discussed in the introduction, there are a number of ‘sub-languages’ one must be familiar with in any setting —e.g., function symbols and types (classifications) and their respective operations— and so a prime goal of our discussions will be to *reduce* the number of distinctions so that we have a *uniform* approach to different aspects of a language.

The goals of the subsections are as follows.

Provide a formalism of the above `Colour` package

1. **What is a language?** Sketch out the English sentences example from above, introducing the notation used for declaring grammars of languages, along with typing contexts.

2. **Signatures** Attempt to extrapolate the key ideas of the previous section; concluding with a a discussion of when contexts constitute packages.

3. **Presentations of Signatures —Π and Σ** The desire to present packages (signatures) *practically* in a uniform notation leads to types that *vary* according to other types and so the constructor Π; then the **(un)bundling problem** is used to motivate the introduction of the Σ type constructor.

Demonstrate the interdefinability of structuring mechanisms

4. **A Whirlwind Tour of Agda** Tersely review the Agda language as a tool supporting the ideas of the previous subsections. In particular, the usual structuring mechanisms found in most settings are discussed —they are records, namespacing modules, and "algebraic datatypes" (grammars in a new setting).

5. **Facets of Structuring Mechanisms** Demonstrate three possible ways to define monoids in Agda and argue their equivalence; thereby, showing that structuring mechanisms are in effect accomplishing the same goal in different ways: They package data along with a particular *usage interface*. As such, it is not unreasonable to seek out a unified notion of **package** —namely, the aforementioned generalised signatures.

Take inspiration from how other DTLs handle packages

6. **Contexts are Promising** Discuss how other dependently-typed languages (DTLs) view contexts and signatures.

7. **Coq Modules as Generalised Signatures** Argue that the notion of generalised signature is promising as the underlying formal definition of packages.

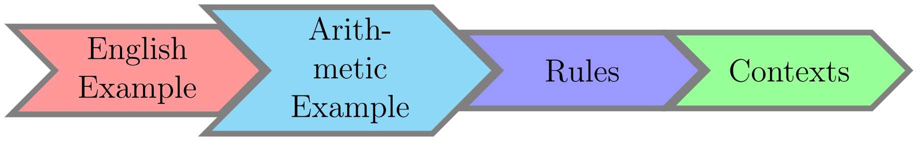
Contributions of the thesis

8. What is the primary problem the thesis aims to address.

9. What are the outcomes of the thesis effort.

## 2.1 What is a language?

In this section, we introduce two languages in preparation for the terminology and ideas of the next section. The first language, *Madlips*, will only be discussed briefly and is mentioned due to its inherit accessibility, thereby avoiding unnecessary domain specific clutter and making definitions clearer. The plan for this section is loosely summarised by the following diagram.



**Madlips**[1]: Simple English sentences have the form subject-verb-object such as *"Jim ate the apple"*. To *mindlessly* produce such sentences, one must produce a subject, then a verb, then an object —all from given lists of possibilities. A convenient notation to describe a language is its *grammar* [**DBLP:journals/iandc/Chomsky59b**; **DBLP:journals/iandc/Chomsky59a**] presented in *Backus-Naur Form* [**DBLP:journals/cj/ChaplinCH73**; **DBLP:conf/aplas/GuoyongDF** **DBLP:journals/bmcbi/LarosBDT11**; **DBLP:journals/cacm/Knuth64a**] as in Figure **??**.

The notation $\tau$ ::= $\mathtt{c}_0$ | $\mathtt{c}_1$ | ... | $\mathtt{c}_n$ defines the name $\tau$ as an alias for the collection of words —also called *strings* or *constructors*— $\mathtt{c}_0$ or $\mathtt{c}_1$ or ... or $\mathtt{c}_n$; that is the bar '|' is read 'or'. The name $\tau$ is also known as a *syntactic category.* For example, in the Madlips grammar, `Subject` is the name of the collection of words *Jim, He,* and *Apple.* A constructor may be followed by words of another collection, which are called *the arguments of the constructor.* For example, the `Object` collection above has a 'The' constructor which must be followed by a word of the `Subject` collection; e.g, `The Apple` is a valid *value* of the `Object` collection, whereas `The` is just an incomplete construction of `Object` words. The last clause of `Object` is just `Subject`: An invisible (unwritten) constructor that takes a value of `Subject` as its argument; e.g., `He` and all other values of `Subject` are also values of the `Object` collection. Similarly, the `Sentence` collection consists of one invisible (unwritten) constructor that takes 3 arguments —a subject, a verb, and an object. Below is an example *derivation* of a *sentence*

[1]This is a collection of English sentences that may result from the *lips* of a person who is *mad.* Example phrases include `He Ate The Apple, He Ate Jim,` and `Apple Kissed The Jim` —whereas the first is reasonable, the second is worrisome, and the final phrase is confusing.

```
Subject  ::= Jim | He | Apple
Verb     ::= Ate | Kissed
Object   ::= The Subject | Subject
Sentence ::= Subject Verb Object
```

Figure 2.1: Madlips Grammar

in the *language generated by this grammar*; at each '$\rightarrow$' step, one of the collection names is replaced by one of its constructors until there are no more possible replacements.

Example Derivation

```
   Sentence
→  Subject Verb Object
→  Jim     Verb Object
→  Jim     Ate  Object
→  Jim     Ate  The Subject
→  Jim     Ate  The Apple
```

Similarly, one may form `He Kissed Jim` as well as the meaningless sentence `Apple Kissed He`.

- ◇ The first is vague, the pronoun 'He' does not designate a known person but instead "stands in" for a *variable*, yet unknown, person. As such, the first sentence can be assigned a meaning once we have a *context* of which pronouns refer to which people.
- ◇ The second just doesn't make sense. Sometimes nonsensical sentences can be avoided by restructuring the grammar, say, by introducing auxiliary syntactic categories. A more general solution is to introduce *judgement rules* that characterise the subset of sentences that are sensible.

We will return to the notions of *context* and *judgement* after the next example language.

**Freshmen**: Introductory computing classes are generally interested in arithmetic that involves both numeric and truth values —also known as *Boolean values*. We can capture some of their ideas with the following grammar.

Freshmen Grammar

```
Term ::= Zero | Succ Term | Term + Term | True | False | Term ≈ Term
```

- ◇ Unlike the previous grammar, instead of `+ Term Term` to declare a constructor '+' that takes two `Term` values, we write the operation `_+_` *infix*[2], in the middle, since that is a common convention for such an operation. Likewise, `Term` ≈ `Term` specifies a constructor `_≈_` that takes two term values.

Example terms include the numbers `Zero`, `Succ Zero`, and `Succ Succ Zero` —which denote 0, 1 (the successor of zero), and 2 (the successor of the successor of zero). The

[2]It is common to use underscores "_" to denote the *position* of arguments to constructions that do not appear first in a term. For example, one writes `if_then_else_` to indicate that we have a construction that takes *three* arguments, as indicated by the number of underscores; whence in a term such as `if` $x$ `then` $y$ `else` $z$ it is understood that we have the construction `if_then_else_` applied to the arguments $x$, $y$, and $z$.

sensible Booleans terms `True ≈ False` and `True` are also possible —regardless of *how true* they may be. However, the nonsensical terms `True + False` and `Zero ≈ True` are also possible. As mentioned earlier, judgement rules can be used to characterise the sensible terms: The relationship "term $t$ is an element of kind $\tau$", written `t : τ` is defined by (1) introducing a new syntactic category (called "types") to 'tag' terms with the kind of elements they denote, and (2) declaring the conditions under which the relationship is true.

**Types for Freshmen**

```
Type ::= Number | Boolean
```

**Judgement Rules**

$$\frac{}{\texttt{Zero} : \texttt{Number}} \qquad \frac{t : \texttt{Number}}{\texttt{Succ}\, t : \texttt{Number}} \qquad \frac{s : \texttt{Number} \quad t : \texttt{Number}}{s + t : \texttt{Number}} \qquad \frac{}{\texttt{True} : \texttt{Boolean}}$$

$$\frac{}{\texttt{False} : \texttt{Boolean}} \qquad \frac{s : \texttt{Number} \quad t : \texttt{Number}}{s \approx t : \texttt{Boolean}} \qquad \frac{s : \texttt{Boolean} \quad t : \texttt{Boolean}}{s \approx t : \texttt{Boolean}}$$

A rule $\frac{premises}{conclusion}$ means "if the top parts are all true, then the bottom part is also true"; some rules have no premises and so their conclusions are unconditionally true. That these are *judgement rules* means that a particular instance of the relationship `t : τ` is true if and only if it is the conclusion of 'repeatedly stacking' these rules on each other. For example, below we have a *derivation tree* that allows us to conclude the sentence `Zero ≈ Succ Zero` is a Boolean term —regardless of *how true* the equality may be. Such trees are both read and written from the *bottom to the top*, where each horizontal line is an invocation of one of the judgement rules from above, until there are no more possible rules to apply.

$$\frac{\dfrac{}{\texttt{Zero} : \texttt{Number}} \qquad \dfrac{\dfrac{}{\texttt{Zero} : \texttt{Number}}}{\texttt{Succ}\,\texttt{Zero} : \texttt{Number}}}{\texttt{Zero} \approx (\texttt{Succ}\,\texttt{Zero}) : \texttt{Boolean}}$$

This solves the problem of nonsensical terms; for example, `True + Zero` *cannot be assigned* a type since the judgement rule involving `_+_` requires both its arguments to be numbers. As such, **consideration is moved from raw terms, to typeable terms.** The types can be interpreted as *well-definedness constraints* on the constructions of terms. Alternatively, types can be considered as **abstract interpreters** in that, say, we may not know the exact *value* of `s + t` but we know that it is a `Number` *provided* both `s` and `t` are numbers; whereas we know nothing about `Zero + False`.

| Concept | Intended Interpretation |
|---|---|
| type | a collection of things |
| term | a particular one of those things |
| $x : \tau$ | the declaration that $x$ is indeed within collection $\tau$ |

There is one remaining ingredient we have yet to transfer over from the Madlips setting: Pronouns, or *variables*, which "stand in" for "yet unknown" values of a particular type. Since a variable, say, $x$, is a stand-in value, a term such as $x$ `+ Zero` has the `Number` type *provided* the variable $x$ is known, in a *context*, to be of type `Number` as well. As such, in the presence of variables, the typing relation `_:_` must be extended to, say, `_⊢_:_` so that we have **typed terms in a context**.

$$\Gamma \vdash t : \tau \qquad \equiv \qquad \text{“}\textit{In the context } \Gamma\textit{, term } t \textit{ has type } \tau\text{”}$$

A *context*, denoted $\Gamma$, is simply a list of associations: In Madlips, a context associates pronouns with the names of people they refer to; in Freshmen, a context associates variables with their types. For example, $\Gamma : \texttt{Variable} \to \texttt{Type}; \Gamma(x) = \texttt{Number}$ associates the `Number` type to every variable. In general, a context only needs to mention the pronouns (variables) used in a sentence (term) for the sentence (term) to be understood, and so it may be **presented** as a set of pairs $\Gamma = \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\}$ *with* the understanding that $\Gamma(x_i) = \tau_i$. However, since we want to *treat* each association $(x_i, \tau_i)$ as saying "$x_i$ has type $\tau_i$", it is common to present the **tuples** in the form $x_i : \tau_i$ —that is, the colon ':' is **overloaded** for denoting tuples in contexts and for denoting typing relationships.

**Extending Freshmen with Variables**

```
Term     ::= ··· | Variable
Variable ::= x | y | z
```

We have one new rule to type variables, which makes use of the underlying context.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

All previous rules now must now additionally keep track of the context; e.g., the `_+_` rule becomes:

$$\frac{\Gamma \vdash s : \texttt{Number} \quad \Gamma \vdash t : \texttt{Number}}{\Gamma \vdash s \;\texttt{+}\; t : \texttt{Number}}$$

We may now derive $x$ : `Number` ⊢ $x$ `+ Zero` : `Number` but cannot complete the senseless phrase $x$ : `Boolean` ⊢ $x$ `+ Zero` : `???`. *That is, the same terms may be typeable in some contexts but not in others.*

Before we move on, it is interesting to note that contexts can themselves be presented with a grammar —as shown below, where constructors ',' and ':' each take two arguments and are written infix; i.e., instead of the usual `, arg`$_1$ `arg`$_1$ we write `arg`$_1$ `, arg`$_2$. Contexts are *well-formed* when variables are associated at most one type; i.e., when contexts *represent* 'partial functions'.

Grammar for Contexts

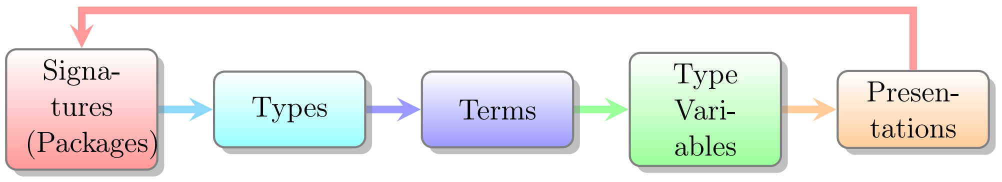
```
Context     ::= ∅ | Association, Context
Association ::= Variable : Type
```

Finally, it is interesting to observe that the addition of variables results in a an interesting correspondence: **Terms in context are functions of their variables**. More precisely, if there is a method $[\![\_]\!]$ that *interprets* type names $\tau$ as actual sets $[\![\tau]\!]$ and terms $\mathtt{t} : \tau$ as *values* of those sets $[\![\mathtt{t}]\!] : [\![\tau]\!]$, then a **term** in context $\mathtt{x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n \vdash \mathtt{t} : \tau$ corresponds to the **function** $f : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]; f(x_1, \ldots, x_n) = [\![t]\!]$. *That is, terms in context model parameterisation **without** speaking of sets and functions.* ( Conversely, *functions* $A \to B$ "are" *elements* of $B$ *in a context* $A$. )

As mentioned in the introduction, we want to treat packages as the central structure for compound computations. To this aim, we have the approximation: **Parameterised packages are terms in context.**

## 2.2 Signatures

The languages of the previous section can be organised into *signatures*, which define interfaces in computing since they consist of the *names* of the types of data as well as the *names* of operations on the types —there are only symbolic names, not implementations. The purpose of this section is to organise the ideas presented in the previous section —shown again in the figure below— in a refinement-style so that the resulting formal definition permits the presentation of packages given in the first subsection above.



**Signatures** are tuples $\Sigma = (\mathcal{S}, \mathcal{F}, src, tgt)$ consisting of

◇ a set $\mathcal{S}$ of *sorts* —the names of types—,

- ◇ a set $\mathcal{F}$ of *function symbols*, and
- ◇ two mappings `src` $: \mathcal{F} \to$ `List` $\mathcal{S}$ and `tgt` $: \mathcal{F} \to \mathcal{S}$ that associate a list[3] of *source sorts* and a *target sort* with a given function symbol.

**Signatures generalise graphical sketches**

*Unary Signatures* have only one source sort for each function symbol —i.e., the length of `src` $f$ is always 1— and so are just graphs.

| Signatures | $\approx$ | Graphs |
|---|---|---|
| Sorts | | "dots on a page", Nodes, Vertices |
| Function symbols | | "lines between the dots", Edges, Tentacles |

**Typing** the symbols of a signature as follows[4] lets us treat signatures as general forms of 'type theories' since we may speak of 'typed terms'.

$$f : s_1 \times \cdots \times s_n \to t \qquad \equiv \qquad \texttt{src}\, f = [s_1, \ldots, s_n] \;\wedge\; \texttt{tgt}\, f = t$$

Moreover, we regain the *typing judgements* of the previous section by introducing a grammar for *terms*. Given a set $\mathcal{V}$ of **variables**, we may define **terms** with the following grammar.

**Grammar for Arbitrary Terms**

```
Term ::= x                -- A variable; an element of 𝒱
       | f t₁ t₂ ... tₙ    -- A function symbol f of ℱ taking n sorts
                          --    where each tᵢ is a Term
```

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad\qquad \frac{\Gamma \vdash t_1 : \tau_n \quad \ldots \quad \Gamma \vdash t_n : \tau_n \qquad f : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash \texttt{f}\, t_1\, t_2 \ldots t_n : \tau}$$

Figure 2.2: Signature Typing

As discussed in the previous section, variables are *not* necessary and if they are *not* permitted, we omit the first clause of `Term` and only use the second typing rule —we also drop the contexts since there would be no variables for which variable-type associations must be remembered. Without variables, the resulting terms are called *ground terms*. Since terms are defined recursively, inductively, the set of ground terms is non-empty precisely when at least one function symbol `c` needs no arguments, in which case we say `c` is a *constant symbol* and make the following abbreviation:

$$c : \tau \qquad \equiv \qquad \texttt{src}\, c = [\,] \;\wedge\; \texttt{tgt}\, c = \tau$$

[3] We write `List X` for the type of lists with values from `X`. The empty list is written `[]` and `[x₁, x₂, ..., xₙ]` denotes the list of $n$ elements $\texttt{x}_i$ from `X`; one says $n$ is the *length* of the list.

[4] The wedge symbol '$\wedge$' is read "and"; e.g., $p \wedge q$ is read *"p and q are true"*.

Alternatively, the abbreviation $\tau_1 \times \cdots \times \tau_n \to \tau$ is written as just $\tau$ *when* $n = 0$.

How do we actually **present** a signature?

**Brute force** Recall the Freshmen language, we can present an *approximation*[5] of it as signature by providing the necessary components $\mathcal{S}$, $\mathcal{F}$, `src`, and `tgt` as follows —where, for brevity, we write $\mathcal{B}$ and $\mathcal{N}$ instead of `Boolean` and `Number`.

$$\mathcal{S} = \{\texttt{Number, Boolean}\}$$
$$\mathcal{F} = \{\texttt{Zero, Succ, Plus, True, False, Equal}\}$$

| *op* | Zero | Succ | True | False | _+_ | _≈_ |
|---|---|---|---|---|---|---|
| src *op* | [] | [$\mathcal{N}$] | [] | [] | [$\mathcal{N}$, $\mathcal{N}$] | [$\mathcal{N}$, $\mathcal{N}$] |
| tgt *op* | $\mathcal{N}$ | $\mathcal{N}$ | $\mathcal{B}$ | $\mathcal{B}$ | $\mathcal{N}$ | $\mathcal{B}$ |

( For each choice of *op* in the first line, `src op` is defined by the corresponding column of the second line; likewise for `tgt op`. )

This is however rather **clumsy** and not that clear. We may collapse the `src, tgt` definitions into the _:_→_ relation defined above; i.e., replacing *two* definition declarations `src Zero = []` $\wedge$ `tgt Zero = Number` by *one* definition declaration `Zero : Number`. However, function symbol names are still repeated twice: Once in the definition of $\mathcal{F}$ and once in the definition of _:_→_; the latter mentions all the names of $\mathcal{F}$ and so $\mathcal{F}$ may be inferred from the typing relationships. We are left with two declarations: The sorts $\mathcal{S}$ and the typing declarations. However, the set $\mathcal{S}$ only serves to declare its elements as sort symbols; if we use a relationship _: `Type` defined by $\tau$ : `Type` $\equiv \tau \in \mathcal{S}$, then the sort symbols can also be introduced by seemingly similar 'typing declarations'. With this approach, Freshmen can be introduced more naturally[6] as follows.

Freshmen as a Generalised Signature

```
Number  : Type
Boolean : Type

Zero : Number
Succ : Number → Number
_+_  : Number × Number → Number

True  : Boolean
False : Boolean
_≈_   : Number × Number → Boolean
```

[5]This is an approximation since we have constrained the equality construction, _≈_, to take *only* numeric arguments; whereas the original Freshmen allowed both numbers and Booleans as arguments to equality *provided* the arguments have the *same type*. We shall return to this issue later when discussing *type variables*.

[6]It is important to note that there are three relations here with ':' in their name —_:`Type`, _:_→_, and _:_ for constant-typing. See Table 2.1.

What a twist: **Generalised signatures are contexts!** That is, a sequence of name-type associations. More precisely, with the relation `package_` defined below, we can characterise packages as the contexts whose earlier elements allow their later elements to be typeable. For example, the context `S : Type; x : S` can be proven to be package whereas the context `S : Type; x : Q` cannot —it has the 'global name' $Q$.

**Rules for determining when a signature is a package**

A package is a context where later names' types may refer to earlier names.

Given a set $Name$ for variable names and context $\Gamma$, let $FName_\Gamma$ denote the values of $Name$ that do not occur as names in context $\Gamma$ —these are the "fresh names for context $\Gamma$".

$$\frac{}{\mathsf{package}\,\emptyset} \qquad \frac{\mathsf{package}\,\Gamma \qquad \tau \in FName_\Gamma}{\mathsf{package}\,(\Gamma, \tau : \mathsf{Type})}$$

$$\frac{\mathsf{package}\,\Gamma \qquad f \in FName_\Gamma \qquad \Gamma \vdash \tau_i : \mathsf{Type} \ \text{ for each } \tau_i}{\mathsf{package}\,(\Gamma, f : \tau_1 \times \cdots \times \tau_n \to \tau_{n+1})}$$

By using $FName_\Gamma$, names are declared at most once in a context.

Below is an example derivation demonstrating that the context $\mathcal{N}$ `: Type,` $\mathcal{B}$ `: Type, z :` $\mathcal{N}$`, s :` $\mathcal{N} \to \mathcal{N}$ (an initial segment of Freshmen) is actually a package by taking $Name = \{\mathcal{N}, \mathcal{B}, s, z\}$.

$$\dfrac{\dfrac{}{\emptyset, \mathcal{N} : \mathsf{Type}, \mathcal{B} : \mathsf{Type}, z : \mathcal{N} \vdash \mathcal{N} : \mathsf{Type}} \qquad \dfrac{}{s \in FName} \qquad \dfrac{\dfrac{}{\emptyset, \mathcal{N} : \mathsf{Type}, \mathcal{B} : \mathsf{Type} \vdash \mathcal{N} : \mathsf{Type}} \qquad \dfrac{}{z \in FName} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{package}\,\emptyset}}{\mathsf{package}\,(\emptyset, \mathcal{N} : \mathsf{Type})}}{\mathsf{package}\,(\emptyset, \mathcal{N} : \mathsf{Type}, \mathcal{B} : \mathsf{Type})}}{\mathsf{package}\,(\emptyset, \mathcal{N} : \mathsf{Type}, \mathcal{B} : \mathsf{Type}, z : \mathcal{N})}}{\mathsf{package}\,(\emptyset, \mathcal{N} : \mathsf{Type}, \mathcal{B} : \mathsf{Type}, z : \mathcal{N}, s : \mathcal{N} \to \mathcal{N})}$$

It is important to pause and realise that there are **three relations with ':' in their name** —which may include spaces as part of their names.

| | | | |
|---|---|---|---|
| Function symbol to sort *adjacency* | $f : s_1 \times \cdots \times s_n \to s$ | $\equiv$ | $\mathsf{src}\, f = [s_1, \ldots, s_n] \wedge \mathsf{tgt}\, f = s$ |
| Sort symbol *membership* | $s : \mathsf{Type}$ | $\equiv$ | $s \in \mathcal{S}$ |
| *Pair formation* within contexts $\Gamma$ | $x : t$ | $\equiv$ | $(x, t)$ |

Three "typing" relations

Consequently, we have stumbled upon a grammar `TYPE` for types —called the *types for signature* $\Sigma$ over a collection of names $\mathcal{V}$.

**Induced Grammar for Types**

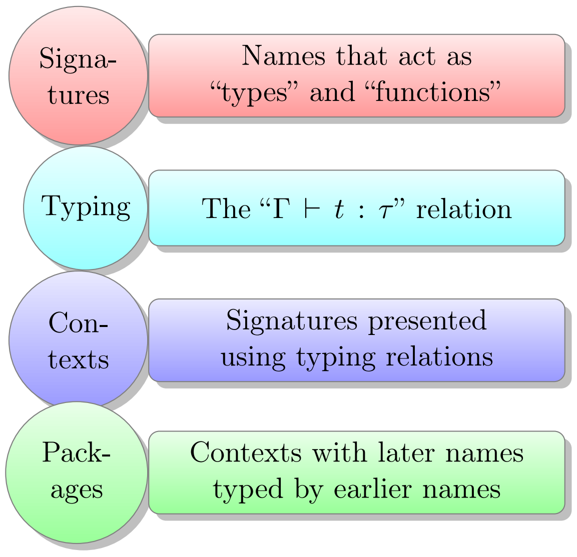
```
TYPE ::= Type               -- An opaque symbol; “the type of types”
      |  τ                  -- τ is a sort symbol; a value of 𝒮
      |  x                  -- A variable; an element of 𝒱
      |  TYPE [→] TYPE       -- _→_ takes two TYPE arguments
      |  TYPE [×] TYPE |  𝟙 -- “product types”
```

The type $\mathbb{1}$ is used for constants: With this grammar a constant $c : \tau$ would have type `c` $: \mathbb{1} \to \tau$. The symbol $\mathbb{1}$ is used simply to indicate that the function symbol `c` takes no arguments. The introduction of $\mathbb{1}$ saves us from having to include the constant-typing relationship defined above —namely, `c` $: \tau \quad\equiv\quad$ `src c = []` $\wedge$ `tgt c =` $\tau$.

We may now form types $\alpha \to \beta$ and $\alpha \times \beta$ but there is no way for the type $\beta$ to depend on the type $\alpha$. In particular, recall that in Freshmen we wanted to have `s` $\approx$ `t` to be a well-formed term of type `Boolean` *provided* `s` and `t` have the *same* type, either `Number` or `Boolean`. That is, `_`$\approx$`_` wants to have *both* `Number` $\times$ `Number` $\to$ `Boolean` *and* `Boolean` $\times$ `Boolean` $\to$ `Boolean` as types —since it is reasonable to compare either numbers *or* truth values for equality. But a function symbol can have only *one* type —since `src` and `tgt` are (deterministic) functions. If we had access to variables which stand-in for types, we could type equality as $\alpha \times \alpha \to$ `Boolean` *for any type* $\alpha$.

$$\frac{}{\alpha : \texttt{Type} \quad\vdash\quad \_\approx\_ : \alpha \times \alpha \to \texttt{Boolean}}$$

Even though types *constrain* terms, there seems to be a subtle repetition: The `TYPE` grammar resembles the `Term` grammar. In fact, if we pretend `Type`, $\mathbb{1}$, `_`$\times$`_`, `_`$\to$`_` *are* function symbols, then `TYPE` is subsumed by `Term`. Hence, we may conflate the two into one declaration to obtain *dependently-typed terms* —a concern which we will return to at a later time. For now, we may summarise our progress with the following figure.


Signa-tures
Names that act as “types” and “functions”
Typing
The “Γ ⊢ t : τ” relation
Con-texts
Signatures presented using typing relations
Pack-ages
Contexts with later names typed by earlier names

## 2.3 Presentations of Signatures —Π and Σ

Since a signature's types also have a grammar, we can present a signature in the natural style of "name : type-term" pairs. That is, a signature may be presented as a context; i.e., sequence of declarations $\delta_0$, $\delta_1$, ..., $\delta_n$ *such that* each $\delta_i$ is of the form `name`$_i$ `:` `type`$_i$ where *name*$_i$ are unique names but *type*$_i$ are **terms** from the `TYPE` grammar. For example, the above presentation of Freshmen is a context from which we regain a signature $\Sigma = (\mathcal{S}, \mathcal{F}, src, tgt)$ where:

- ◇ $\mathcal{S}$ is all of the *name*$_i$ where *type*$_i$ is `Type`;
- ◇ $\mathcal{F}$ is the remaining *name*$_i$ symbols;
- ◇ `src, tgt` are defined by the following equations, where the right side, involving `_:_→_` and `_:_`, are given in the context of $\delta_i$.

  $$\begin{array}{llll} \texttt{src}\, f = [\tau_1, \ldots, \tau_n] & \wedge \quad \texttt{tgt}\, f = \tau & \equiv & f : \tau_1 \times \cdots \times \tau_n \to \tau \\ \texttt{src}\, f = [] & \wedge \quad \texttt{tgt}\, f = \tau & \equiv & f : \tau \end{array}$$

  These equations ensure `src, tgt` are functions *provided* each name occurs at most once as the name part of a declaration.

This is one of the first instances of a syntax-semantics relationship: **A context is a syntactic representation of a (generalised) signature**. However, with a bit of experimentation one quickly finds that the syntax is "too powerful": There are contexts that do *not* denote signatures. Consider the following grammar which models 'smart' people and their phone numbers. Observe that the 'smartness' of a person *varies* according to their location; for example, in, say, a school setting we have 'book smart' people whereas in the city we have 'street smart' people and, say, in front of a television we have 'no smart' people. Moreover, the function symbol `call` for obtaining the phone number of a 'smart person' must necessarily have a variable that accounts for how the smart type *depends* on location. However, if variables are not permitted, then `call` cannot have a type which is unreasonable. It is a well-defined context, but it does not denote a signature.

Calling-smart-people Context

```
Location : Type

School   : Location
Street   : Location
TV       : Location

Smart    : Location → Type

Phone    : Type
call     : Smart ℓ → Phone   -- A variable?!
```

The first problem, the type of `Smart`, is easily rectified: The sorts $\mathcal{S}$ are now *all* names in the context that *conclude* with `Type` or that *conclude* with some $\tau$ that has type `Type`. Sorts now may *vary* or *depend* on other sorts.

The second problem, the type of `call`, requires the introduction of a new[7] type operation. The operation $\Pi\_:\_ \bullet \_$ will permit us to type function symbols that have variables in their types even when there is no variable collection $\mathcal{V}$.

Dependent Function Type

$$\Pi\, a : A \;\bullet\; B\, a \qquad \equiv \qquad \text{“Values of } \textit{type } B\,a\text{, for each value } a \text{ of type } A\text{”}$$

An element of $\Pi\, a : A \bullet B\, a$ is a function $f$ which assigns to each $a : A$ an element of $B\, a$. Such methods $f$ are *choice functions*: For every $a$, there is a collection $B\, a$, and $f\, a$ picks out a particular $b$ in $a$'s associated collection.

The type of `call` is now $\Pi\ \ell$ : `Location` $\bullet$ `(Smart` $\ell \to$ `Phone)`. That is, *given* any location $\ell$, `call` $\ell$ specialises to a function symbol of type `Smart` $\ell \to$ `Phone`, then given any “smart person $s$ in location $\ell$”, `call` $\ell$ `s` would be their phone number. Interestingly, if $s$ is a street-smart person then `call School s` is *ill-typed*: The type of `s` must be `Smart School` not `Smart Street`. Hence, later inputs may be constrained by earlier inputs. This is a new feature that simple signatures did not have.

Before extending the previous definition of signatures, there is a practical subtlety to consider. Suppose we want to talk about smart people *regardless* of their location, how would you express such a type? The type of `call : (`$\Pi\ l$ : `Location` $\bullet$ `Smart` $l \to$ `Phone)` reads: *After picking a particular location $\ell$, you may get the phone numbers of the smart people at that location.* More specifically, $\Pi\ \ell$ : `Location` $\bullet$ `Smart` $\ell$ is the type of smart people **at a particular** location $\ell$. Since, in this case, we do not care about locations, we would like to simply pick a person who is located **somewhere**. The ability to “bundle away” a varying feature of a type, instead of fixing it as a particular value, is known as the **(un)bundling problem**[8]. It is addressed by introducing a new[9] type operator $\Sigma\_:\_ \bullet \_$ —the symbol ‘$\Sigma$’ is conventionally used both for the name of signatures and for this new type operator.

[7]Those familiar with set theory may remark that dependent types are not *necessary* in the presence of power sets: Instead of a *single* name `call`, one uses a (possibly infinite) *family of names* $\mathsf{call}_\ell$ for each possible name $\ell$. Even though power sets are not present in our setting, dependent types provide a natural and elegant approach to *indexed types* in lieu of an encoding in terms of *families of sets or operations*. Moreover, an encoding *hides* essential features of an idea such as dual concepts: $\Sigma$ and $\Pi$ are ‘adjoint functors’. Even more surprising, working with $\Sigma$ and $\Pi$ leads one to interpret “propositions as types” with predicate logic quantifiers $\forall/\exists$ encoded via dependent types $\Pi/\Sigma$; whence the slogan “Programming $\approx$ Proving”.

[8]The initiated may recognise this problem as identifying the relationship between *slice categories* $\mathcal{C}/A$ whose objects are $A$-indexed families and *arrow categories* $\mathcal{C}^{\to}$ whose objects are *all* the $A$-indexed families *for all* possible $A$. In particular, identifying the relationship between the categorial transformations $\_/A$ and $\_^{\to}$ —for which there is a non-full inclusion from the former to the latter, which we call “$\Sigma$-padding”.

[9]The $\Sigma$-types denote disjoint unions and are sometimes written as $\coprod$ —the ‘dual’ symbol to $\Pi$.

| | |
|---|---|
| `Π ℓ : Location • Smart ℓ` | Pick a location, then pick a person |
| `Σ ℓ : Location • Smart ℓ` | Pick a person, who is located *somewhere* |
| `Π a : A • B a` | Pick a value `a : A`, to get `B a` values |
| `Σ a : A • B a` | Pick a value `b : B a`, which is tagged by *some* `a : A` |

**Dependent Product Type**

$\Sigma\, a : A \bullet B\, a \quad\equiv\quad$ "The type of pairs $(a, b)$ where $a : A$ and $b$ is a value of *type* $B\,a$"

An element of $\Sigma\, a : A \bullet B\, a$ is a pair $(a, b)$ of an element $a : A$ along with an element $b : B\, a$. Such pairs are *tagged values*: We have values $b$ which are 'tagged' by the collection-*index* $a$ with which they are associated.

The type operator `_→_` did not accommodate dependence but Π does; indeed if $B$ does not depend on values of type $A$, then $\Pi a : A \bullet B$ is just `A → B`. Likewise, Σ generalises `_×_`.

**Abbreviations**

Provided $B$ is a type that does not vary,

$$\begin{aligned} A \to B &\equiv \Pi\, x : A \bullet B \\ A \times B &\equiv \Sigma\, x : A \bullet B \end{aligned}$$

Since $\Pi/\Sigma$ are the *varying* generalisations of $\to/\times$, sometimes $\Pi/\Sigma$ are written as $(a : A) \to B\, a$ and $(a : A) \times B\, a$, respectively.

Before returning to the task of defining signatures, let us present a number of examples to showcase the differences between dependent and non-dependent types.

**Example 1: People and their birthdays**

Let `Birthday` : `Weekday` → `Type` denote the collection of all people who have a birthday on a given weekday. One says, `Birthday` *is the collection of all people,* ***indexed*** *by their birth day of the week.* Moreover, let `People` denote the collection of all people in the world.

**$\Pi\, d$ : `Weekday` • `Birthday` $d$ is the type of *functions* that given any weekday $d$, yield a person whose birthday is on that weekday.**

Example functions in this type are $f$ and $g$ below…

```
f Monday  = Jim
f Tuesday = Alice

g Monday  = Mark
g Tuesday = Alice
```

… *provided* we live in a tiny world consisting of three people and only two weekdays.

| Person | Birthday |
|---|---|
| Jim | Monday |
| Alice | Tuesday |
| Mark | Monday |

In contrast, `Weekday` → `People` is the collection of functions associating people to weekdays —no constraints whatsoever. E.g., `f d = Jim` is the function that associates `Jim` to every weekday `d`.

**$\Sigma\, d$ : `Weekday` • `Birthday` $d$ is the type of *pairs* $(d, p)$ of a weekday $d$ and a person whose birthday is that weekday.**

Below are two values of this type (✓) and a non-value (×). The third one is a pair $(d, p)$ where $d$ is the weekday `Tuesday` and so the $p$ must be *some* person born on that day, and `Mark` is not such a person in our tiny world.

```
✓ (Monday, Jim)
✓ (Tuesday, Alice)
× (Tuesday, Mark)
```

In contrast, `Weekday` × `People` is the collection of pairs $(w, p)$ of weekdays and people —no constraints whatsoever. E.g., `(Tuesday, Mark)` is a valid such value.

**Example 2: English words and their lengths**

Let $\texttt{English}_{\leq n}$ denote the collection of all English worlds that have at most $n$ letters; let `English` denote *all* English words.

**$\Pi\, n : \mathbb{N} \bullet \texttt{English}_{\leq n}$ is the type of *functions* that given a length $n$, yield a word of that length.**

Below is part of a such a function `f`.

```
       f 0 = ""   -- The empty word
       f 1 = "a"  -- The indefinite article
       f 2 = "to"
       f 3 = "the"
       f 4 = "more"
       ...
```

In contrast, an $f : \mathbb{N} \to \texttt{English}$ is just a list of English words with the $i$-th element in the list being $f\, i$.

**$\Sigma\, n : \mathbb{N} \bullet \texttt{English}_{\leq n}$ is the type of *values* $(n, w)$ where $n$ is a number and $w$ is an English word of that length.**

For instance, `(5, "hello")` is an example such value; whereas `(2, "height")` is not such a value —since the length of `"height"` is *not* 2.

In contrast, $\mathbb{N} \times \texttt{English}$ is any number-word pair, such as `(12, "hi")`.

*Notice that dependent types may **encode properties** of values.*

**Example 3: "All errors are type errors"**

Suppose `get i xs` is the $i$-th element in a list `xs = [x₀, x₁, …, xₙ]`, what is the type of such a method `get`?

Using `get : Lists → ℕ → Value` will allow us to write `get [x₁, x₂] 44` which makes no sense: There is no 44-th element in that 2-element list! Hence, the `get` operation must constrain its numeric argument to be at most the length of its list argument. That is, `get : (Π (xs : Lists) • ℕ< (length xs) → Value)` where `ℕ< n` is the collection of numbers less than $n$. *Now the previous call, `get [x₁, x₂] 44` does not need to make sense since it is ill-typed*: The second argument does not match the required constraining type.

In fact, when we speak of lists we implicitly have a notion of the kind of value type they contain. As such, we should write `List X` for the type of lists with elements drawn from type `X`. Then what is the type of `List`? It is simply `Type → Type`. With this form, `get` has the type `Π X : Type • Π xs : List X • ℕ< (length xs) → X`.

Interestingly, lists of a particular length are known as *vectors*. The type of which is denoted `Vec X n`; this is a type that is *indexed* by *both* another *type* `X` and an *expression* `n`. Of-course `Vec : Type → ℕ → Type` and, with vectors, `get` may be typed `Π X : Type • Π n : ℕ • Vec X n → ℕ< n → X`; in-particular notice that the *external computation* `length xs` in the previous typing of `get` is replaced by the *intrinsic index* `n`; that is, **dependent types allow us to encode properties of elements at the type level!**

Anyhow, back to the task as hand —defining signatures (packages).

Given two collections of "names" $\mathcal{V}$ and "base symbols" $\mathcal{B}$, we may form the collection of generalised terms as follows —for brevity we ignore the unit type $\mathbb{1}$.

Generalised Terms

```
Term ::= x              -- A "variable"; a value of 𝒱
       | β              -- A "base symbol; a value of ℬ
       | Type           -- The type of types
       -- For previously constructed types τ and τ′,
       -- previously constructed terms tᵢ,
       -- and variable x:
       | (Π x : τ • τ′) | (λ x : τ • t)          |  t₁ t₂
       | (Σ x : τ • τ′) | let (t₁, t₂) ≔ t₃ in t₄ | (t₁, t₂)
```

This collection constructs a number of different kinds of things: If `t : τ` and `τ : Type` we refer to `t` as an **expression**, to $\tau$ as a **type**, and to `Type` as a **kind**. The following table provides an intuitive interpretation of these terms.

**Intended Interpretations of Generalised Terms**

| Symbols | Intended Interpretation |
|---|---|
| `Type` | The type of all types |
| $\mathbb{1}$ | The type with one element; an example of a base symbol |
| $\Pi$ `a : A • B a` | Values of *type* `B a`, for each value `a` of type `A` |
| $\lambda$ `x :` $\tau$ `• t` | The function that takes input `x :` $\tau$ and yields output `t` |
| `f e` | Apply the function `f` on input term `e` |
| $\Sigma$ `a : A • B a` | Pairs `(a, b)` where `a : A` and `b` is a value of *type* `B a` |
| `(x , w)` | A pair of items where the second may depend on the first |
| `let (x, w) ≔` $\beta$ `in e` | Unpack the pair $\beta$ as the pair `(x, t)` for use in term `e` |

**Abbreviations**: Provided $B$ is a type that does not vary,

| Symbol | | Elaboration | Intended Interpretation |
|---|---|---|---|
| $A \to B$ | $\equiv$ | $\Pi\, x : A \bullet B$ | The functions from $A$ to $B$ |
| $A \times B$ | $\equiv$ | $\Sigma\, x : A \bullet B$ | Pairs of values *(a, b)* with $a : A$ and $b : B$ |

The rules below classify the well-formed generalised terms. The rules for $\Pi$ and $\Sigma$ show that they are *families* of types 'indexed' by the first type. The rules only allow the construction of types and variable values, to construct *values of types* we will need some starting base types, whence the upcoming definition.

## Judgements for Generalised Terms

$$\frac{}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}}[\textsc{Type-in-Type}] \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}[\textsc{Variables}]$$

The VARIABLES rule is also known as ASSUMPTION and may be rendered as follows.

$$\frac{}{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_i : \tau_i}[\textsc{Variables}]$$

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

$$\frac{\Gamma, x : \tau \vdash \tau' : \mathsf{Type}}{\Gamma \vdash (\Pi\, x : \tau \bullet \tau') : \mathsf{Type}}[\Pi\text{-}\textsc{Formation}]$$

$$\frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash (\lambda\, x : \tau \bullet t) : (\Pi\, x : \tau \bullet \tau')}[\Pi\text{-}\textsc{Introduction}]$$

$$\frac{\Gamma \vdash \beta : (\Pi\, x : \tau \bullet \tau') \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \beta\, t : \tau'[x \coloneqq t]}[\Pi\text{-}\textsc{Elimination}]$$

The notation $E[x := F]$ means “replace every occurrence of the name $x$ within term $E$ by the term $F$.” This ‘find-and-replace’ operation is formally known as *textual substitution*.

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

$$\frac{\Gamma, x : \tau \vdash \tau' : \mathsf{Type}}{\Gamma \vdash (\Sigma\, x : \tau \bullet \tau') : \mathsf{Type}}[\Sigma\text{-}\textsc{Formation}]$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash t : \tau'[x \coloneqq e]}{\Gamma \vdash (e, t) : (\Sigma\, x : \tau \bullet \tau')}[\Sigma\text{-}\textsc{Introduction}]$$

$$\frac{\Gamma \vdash \beta : (\Sigma\, x : \tau \bullet \tau') \qquad \Gamma, x : \tau, t : \tau' \vdash \gamma : \tau''}{\Gamma \vdash \mathsf{let}\ (x, t) \coloneqq \beta\ \mathsf{in}\ \gamma : \tau''}[\Sigma\text{-}\textsc{Elimination}]$$

Just as $\Sigma$ is the dual to $\Pi$, in some suitable sense, so too the *eliminator* `let` is dual to the *constructor* lambda $\lambda$.

**$\Pi$ and $\Sigma$ together allow the meta-language to be expressed in the object-language**

Recall that a phrase "$\Gamma \vdash t : \tau$" denotes a property that **we** check using day-to-day mathematical logic in conjunction with the provided rules for it. In turn, the property **talks about** terms $t$ and $\tau$ which are related provided assumptions $\Gamma$ are true. In particular, contexts and the entailment relation are *not* expressible as terms of the object language; i.e., they cannot appear in the $t$ nor the $\tau$ positions … that is, until now.

**$\Pi$ types *internalise* contexts**

Contextual information is 'absorbed' as a $\lambda$-term; that is, $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$ is essentially $\vdash (\lambda x_1 : \tau_1 \bullet \cdots \bullet \lambda x_n : \tau_n \bullet t) : (\Pi x_1 : \tau_1 \bullet \cdots \bullet \Pi x_n : \tau_n \bullet \tau)$.
Recall that initially we remarked that terms-in-context are essentially functions *provided* we have some form of semantics operation $[\![\_]\!]$. However, in the presence of $\Pi$ types, terms-in-context correspond to functional terms in the *empty* context. The $\Pi$-Formation rule "explains away" the new $\lambda$-terms using the old familiar notion of contexts.

**$\Sigma$ types *internalise* pairing contexts**

Multiple contexts are 'fused' as a $\Sigma$-type term; that is, *multiple* premises in a judgement rule can be replaced by a *single* premise by repeatedly using $\Sigma$-Formation.

A **Generalised Signature** is a tuple *($\mathcal{B}$, type)* where $\mathcal{B} = [\beta_0, \beta_1, \ldots, \beta_n]$ is an *ordered* list of "base symbols" and `type` $: \mathcal{B} \to$ `Term` associates a generalised term to each base symbol such that $\Gamma_{k-1} \vdash \texttt{type}\, \beta_k : \texttt{Type}$ for each $k : 0..n$, where $\Gamma_k = (\beta_0 : \tau_0, \ldots, \beta_k : \tau_k)$ and $\tau_i = \texttt{type}\, \beta_i$. That is `type` associates to each base symbol a type-term that is well-defined according to the typing rules above for generalised terms and *possibly* making use of previous symbols in the listing. We may now augment the above rule listing so that we can form well-typed *expressions* as well as *terms* using the symbols of $\mathcal{B}$.

**Judgement for Generalised Signatures**

$$\frac{\texttt{type}\, \beta \ = \ \tau}{\Gamma \ \vdash \ \beta : \tau}[\textsc{Base Symbol Introduction}]$$

Crucially, generalised signatures may be presented as a sequence of "symbol : type" pairs where the symbols are unique names and each type is a generalised term. Below is an example similar to the calling-smart-people example from the previous section. In this example, `A` denotes a collection that each member `a : A` of which determines a collection `B a` which each have a 'selected point' `it a : B a`. More concretely, thinking of `A` as the countries in the world from which `B` are the households in each country, then `it` selects a representative member of a household `B a` for each country `a : A`.

```
                                   Pointed Families
  A  : Type
  B  : A → Type
  it : Π a : A • B a
```

This is a generalised signature $(\mathcal{B},\ type)$ where:

| $\mathcal{B}$ | `A` | `B` | `it` |
|---|---|---|---|
| `type` | `Type` | `A → Type` | `Π a : A • B a` |

The $\Gamma_{k-1} \vdash$ `type` $\beta_k$ : `Type` obligations for this example become:

1. $\vdash$ `Type : Type`,
2. `A : Type` $\vdash$ `(A → Type) : Type`, and
3. `A : Type, B : A → Type` $\vdash$ `(Π a : A • B a) : Type`.

The first is just the Type-in-Type rule, the second is a mixture of the Abbreviation and Π-Formation rules; the third one is a mixture of the Π-Formation, Base Symbol Introduction, and Π-Elimination rules. Moreover, notice that `it a` is a valid term *provided* `a : A` as shown in the following derivation.

$$\dfrac{\dfrac{}{a : A \ \vdash\ \texttt{it} : (\Pi\, x : A \bullet Bx)}[\textsc{Symbol Intro}] \qquad \dfrac{}{a : A \ \vdash\ a : A}[\textsc{Variables}]}{a : A \ \vdash\ \texttt{it}\, a \ :\ B\, a}[\Pi\text{-}\textsc{Elim}]$$

Signatures are a staple of computing science since they formalise interfaces and generalise graphs and type theories. Our generalised signatures have been formalised "after the fact" from the creation of the prototype for packages. In the literature, our definition of generalised signatures is essentially a streamlined presentation of Cartmell's *Generalised Algebraic Theories* [**DBLP:journals/apal/Cartmell86**] expect that we do not allow arbitrary equational 'axioms' instead using "name = term" rather than "term = term" axioms which serve as *default implementations* of names. We now turn to extending the current setup to permit optional definitions.

## Permitting Optional Definitions

The example packages from this chapter's introduction, one of which is shown below for convenience, can *almost* be understood as presentations of generalised signatures. What is lacking is the ability for *optional* definitions, as is the case with `violet` and `dark` below.

A dynamical system – Colours

```
Colour : Type
red    : Colour
green  : Colour
blue   : Colour
mix    : Colour ⊠ Colour ⊡→ Colour
violet : Colour
violet = mix green blue
dark   : Colour ⊡→ Colour
dark c = mix c blue
```

Recall that the crucial feature of generalised signatures is that they may be presented as a sequence of *declarations* $\delta_1, \ldots, \delta_n$. When written with multiple lines, the commas are replaced by newlines —as with `Colour` above. Originally, each $\delta_i$ is of the form "*name* : *type*", but above we have a definition for `violet`, so the first step is to redefine *declaration* so that each $\delta_i$ is of the form "$\eta : \tau = d$" where the first term $\tau$ is of type `Type` and the second term $d : \tau$. In the multi-line rendition, $\eta : \tau = d$ occurs as two lines: One with $\eta : \tau$ and one with $\eta = d$; c.f., `violet` above. The only ingredient missing is the variable support in `dark` above: What could $\eta$ `x = d` mean? Since `d` is defined *in the context* of `x` and $\lambda$-terms internalise contexts, as discussed above, we can take $\eta$ `x = d` to be an abbreviation for $\eta$ `= (`$\lambda$ `x :` $\tau$ `• d)` for a suitable type $\tau$.

A **Generalised Signature** is now defined to be a tuple $(\mathcal{B}, \mathit{type}, \mathit{definition})$ where $\mathcal{B} = [\beta_0, \beta_1, \ldots, \beta_n]$ is an *ordered* list of "base symbols", `type` $: \mathcal{B} \to$ `Term` associates a generalised term to each base symbol such that $\Gamma_{k-1} \vdash \tau_k :$ `Type` for each $k : 0..n$, where $\Gamma_k = (\beta_0 : \tau_0, \ldots, \beta_k : \tau_k)$ and $\tau_i =$ `type` $\beta_i$; and `definition` $: \mathcal{B} \to$ `Term` is a partial function associating a term to each symbol name such that the types agree: $\Gamma_{k-1} \vdash$ `definition` $\beta_k :$ `type` $\beta_k$. That is `type` associates to each base symbol a type-term that is well-defined according to the typing rules above for generalised terms and *possibly* making use of previous symbols in the listing. Then `definition` $\beta_k$ *may* provide a description of a value of `type` $\beta_k$.

Crucially, a generalised signature may be presented as a sequence of declarations $\delta_1, \ldots, \delta_n$ where each $\delta_i$ is of the form "*name* : *term* = *term*" where the "= *term*" portion is optional and the names are unique. When presented with multiple lines, we replace commas by newlines, and split "*name* : *type* = *definition*" into two lines: The first being "*name* : *type*" and the second, if any, being "*name* = *definition*". Moreover, `name = (`$\lambda$ `x :` $\tau$ `• e)` is instead simplified to `name x = e`.

**Example 4: Colours as Generalised Signatures**

For example, the `Colours` context above is a generalised signature, as follows —where, for brevity, we write $\mathbf{C}$ in place of `Colour`.

| $\mathcal{B}$ | $\mathbf{C}$ | `Red` | `green` | `blue` | `mix` | `violet` | `dark` |
|---|---|---|---|---|---|---|---|
| `type` | `Type` | $\mathbf{C}$ | $\mathbf{C}$ | $\mathbf{C}$ | $\mathbf{C} \times \mathbf{C} \to \mathbf{C}$ | $\mathbf{C}$ | $\mathbf{C} \to \mathbf{C}$ |
| `definition` | - | - | - | - | - | $\mathsf{mix\,green\,blue}$ | $\lambda c : \mathbf{C} \bullet \mathsf{mix}\, c\, \mathsf{blue}$ |

**Example 5: Disjoin Sums as Generalised Signatures**

The type `X + Y` denotes the collection of values of the form "in left" `inl x` or "in right" `inr y` for all `x : X` and `y : Y`. That is, `X + Y` is the disjoint union of collections `X` and `Y`. Below are "default implementations" for `_+_, inl, inr`; however, there are other ways to encode sum types.

Sums from Σ and 𝔹

```
𝔹               : Type
True            : 𝔹
False           : 𝔹
_if_then_else_ : Π A : Type • 𝔹 → A → A → A

_+_ : Type → Type → Type
_+_ X Y  =  Σ tag : 𝔹 •  Type if tag then X else Y

inl : Π X : Type • X → 𝔹 × X
inl X x = (True, x)

inr : Π Y : Type • Y → 𝔹 × Y
inr Y y = (False, y)
```

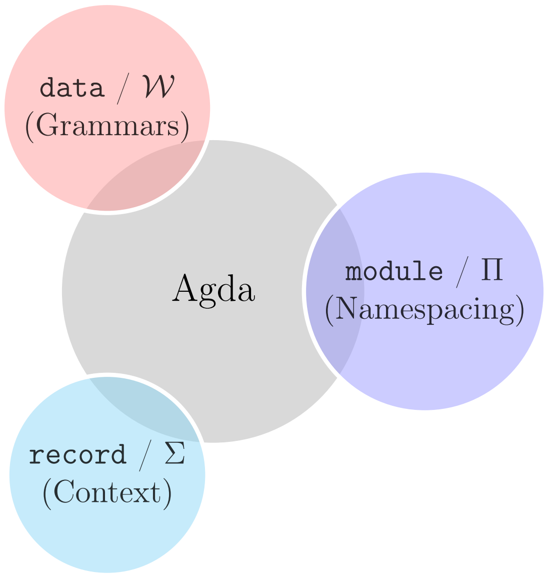
Of course contexts now associate *both* a type and an optional definition with a given name, and so $\Gamma : \mathsf{Name} \to \mathsf{Term} \times (\mathsf{Term} \cup \{-\})$ where "$-$" denotes "no definition". That is, we essentially have two judgement relations $\Gamma \vdash \eta : \tau$ and $\Gamma \vdash \eta : \tau \coloneqq d$ where the extra information in the second can be dropped to get back to the first relation —c.f., $\coloneqq$-Elimination below. We augment our rules with the following two to accommodate this extended capability.

$$\frac{\Gamma(\eta) = (\tau, d)}{\Gamma \;\vdash\; \eta : \tau \coloneqq d}[\coloneqq\text{-Introduction}] \qquad\qquad \frac{\Gamma \;\vdash\; \eta : \tau \coloneqq d}{\Gamma \;\vdash\; \eta : \tau}[\coloneqq\text{-Elimination}]$$

Readers familiar with elementary computing may note that our contextual presentations, when omitting types, are essentially "JSON objects"; i.e., sequences of key-value pairs where the keys are operation names and the values are term descriptions, possibly the "null" description "$-$".

## 2.4 A Whirlwind Tour of Agda

We have introduced a number of concepts and it can be difficult to keep track of when relationships $\Gamma \vdash t : \tau$ are in-fact derivable. The Agda **why_dependent_types_matter**; **dependent_matching_is_just_K**; **curry_howard**; **agda_plf** programming language will provide us with the expressivity of generalised signatures and it will keep track of contexts $\Gamma$ for us. This section recasts many ideas of the previous sections using Agda notation, and introduces some new ideas. In particular, the 'type of types' `Type` is now cast as a hierarchy of types which can contain types at a 'smaller' level: One writes $\texttt{Set}_i$ to denote the type of types at *level* $i : \mathbb{N}$. This is a technical subtlety and may be ignored; instead treating every occurrence of $\texttt{Set}_i$ as an alias for `Type`.



**Unicode Notation**

Unlike most languages, Agda not only allows arbitrary mixfix Unicode lexemes, identifiers, but their use is encouraged by the community as a whole. Almost anything can be a valid name; e.g., `[]` and `_::_` to denote list constructors —underscores are used to indicate argument positions. Hence it is important to be liberal with whitespace; e.g., `e:`$\tau$ is a valid identifier, whereas `e : `$\tau$ declares term `e` to be of type $\tau$. Agda's Emacs interface allows entering Unicode symbols in traditional LaTeX-style; e.g., `\McN`, `\_7`, `\::`, `\to` are replaced by $\mathcal{N}$, $_7$, ::, $\rightarrow$. Moreover, the Emacs interface allows programming by gradual refinement of incomplete type-correct terms. One uses the "hole" marker `?` as a placeholder that is used to stepwise write a program.

### 2.4.1 Dependent Functions — Π-types

A *Dependent Function type* has those functions whose result *type* depends on the *value* of the argument. If `B` is a type depending on a type `A`, then `(a : A) → B a` is the type of functions `f` mapping arguments `a : A` to values `f a : B a`. Vectors, matrices, sorted lists, and trees of a particular height are all examples of dependent types. One also sees the notations `∀ (a : A) → B a` and `Π a : A • B a` to denote dependent types.

For example, *the* generic identity function takes as *input* a type `X` and returns as *output* a function `X → X`. Here are a number of ways to write it in Agda.

The Identity Function

```
id₀ : (X : Set) → X → X
id₀ X x = x

id₁ id₂ id₃ : (X : Set) → X → X

id₁ X = λ x → x
id₂   = λ X x → x
id₃   = λ (X : Set) (x : X) → x
```

All these functions explicitly require the type `X` when we use them, which is silly since it can be inferred from the element `x`. Curly braces make an argument *implicitly inferred* and so it may be omitted. E.g., the `{X : Set} → ⋯` below lets us make a polymorphic function since `X` can be inferred by inspecting the given arguments. This is akin to informally writing $\mathsf{id}_X$ versus `id`.

Inferring Arguments...

```
id : {X : Set} → X → X
id x = x

sad : ℕ
sad = id₀ ℕ 3

nice : ℕ
nice = id 3
```

...and Explicitly Passsing Implicits

```
explicit : ℕ
explicit = id {ℕ} 3

explicit′ : ℕ
explicit′ = id₀ _ 3


.
```

Notice that we may provide an implicit argument *explicitly* by enclosing the value in braces in its expected position. Values can also be inferred when the `_` pattern is supplied in a value position. Essentially wherever the typechecker can figure out a value —or a type—, we may use `_`. In type declarations, we have a contracted form via ∀ —which is **not** recommended since it slows down typechecking and, more importantly, types *document* our understanding and it's useful to have them explicitly.

In a type, `(a : A)` is called a *telescope* and they can be combined for convenience.

```
   (a₁ : A) → {a₂ : A} → {z : _} → (b : B) → ···
≈  (a₁ {a₂} : A) {z : _} (b : B) → ···
≈  ∀ a₁ {a₂ z} b → ···
```

Agda supports the ∀ and the $(a : A) \to B\, a$ notations for dependent types; the following declaration allows us to use the Π notation.

Π Notation in Agda

```
Π∶• : ∀ {a b} (A : Set a) (B : A → Set b) → Set _
Π∶• A B = (x : A) → B x

infix -666 Π∶•
syntax Π∶• A (λ x → B) = Π x ∶ A • B -- The '∶' is Ghost colon, \:
```

The "`syntax function args = new_notation`" clause treats occurrences of `new_notation` as aliases for proper function calls `f x`$_1$ `x`$_2$ `...` `x`$_n$. The `infix` declaration indicates how complex expressions involving the new notation should be parsed; in this case, the new notation binds less than any operator in Agda.

## 2.4.2 Dependent Datatypes — ADTs

Recall that grammars permit a method to discuss "possible scenarios", such as a verb clause or a noun clause; in programming, it is useful to be able to have 'possible scenarios' and then program by considering each option. For instance, a natural number is either zero or the successor of another number, and a door is either open, closed, or ajar to some degree.

Informal Grammar Notation

```
Door ::= Open | Closed | Ajar ℕ
```

Agda Rendition of Grammars

```
data Door : Set where
    Open   : Door
    Closed : Door
    Ajar   : ℕ → Door
```

While the Agda form looks more verbose, it allows more possibilities that are difficult to express in the informal notation —such as, having *parameterised*[10] languages/types for which

[10] With the "types as languages" view, one may treat a "parameterised type" as a "language with dialects". For instance, instead of a single language `Arabic`, one may have a *family of languages* `Arabic ℓ` that depend on a location ℓ. Then, some words/constructors may be accessible in *any* dialect ℓ, whereas other words can only be expressed in a *particular* dialect. More concretely, we may declare `SalamunAlaykum : ∀ {ℓ} →`

the constructors make words belonging to a *particular* parameter only; the `Vec` example below demonstrates this idea.

Languages, such as C, which do not support such an "algebraic" approach, force you, the user, to actually choose a particular representation —even though, it does not matter, since we only want *a way to speak of* "different cases, with additional information". The above declaration makes a new datatype with three different scenarios: The `Door` collection has the values `Open`, `Closed`, and `Ajar n` where `n` is any number —so that `Ajar 10` and `Ajar 20` are both values of `Door`.

Interpreting the Door Values as Options

```
-- Using Door to model getting values from a type X.
-- If the door is open, we get the "yes" value
-- If the door is closed, we get the "no" value
-- If the door is ajar to a degree n, obtain the "jump n" X value.
walk : {X : Type} (yes no : X) (jump : ℕ → X) → Door → X
walk yes no jump Open     = yes
walk yes no jump Closed   = no
walk yes no jump (Ajar n) = jump n
```

**What is a constructor?** A grammar defines a language consisting of sentences built from primitive words; a *constructor* is just a word and a word's *meaning* is determined by how it is used —c.f., `walk` above and the `Vec` construction below which gives us a way to talk about lists. The important thing is that a grammar defines languages, via words, without reference to meaning. Programmatically, constructors could be implemented as "(value position, payload data)"; i.e., pairs `(i, args)` where `i` is the position of the constructor in the list of constructors and `args` is a tuple values that it takes; for instance, `Door`'s constructors could be implemented as `(0,()), (1, ()), (2, (n))` for `Open, Closed, Ajar n` where we use `()` to denote "the empty tuple of arguments". The **purpose** of such types is that we have a number of *distinct* scenarios that may contain a 'payload' of additional information about the scenario; it is preferable to have **informative** (typed) names such as `Open` instead of strange-looking pairs `(0, ())`. In case it is not yet clear, unlike functions, a value construction such as `Ajar 10` cannot be simplified any further; just as the pair value `(2, 5)` cannot be simplified any further. Table 2.2 below showcases how many ideas arise from grammars.

Such "enumerated type with payloads" are also known as **algebraic data types** (ADTs). They have as values $\mathtt{C}_i\ \mathtt{x}_1\ \mathtt{x}_2\ \ldots\ \mathtt{x}_n$, a constructor $\mathtt{C}_i$ with payload values $\mathtt{x}_i$. Functions are then defined by 'pattern matching' on the possible ways to *construct* values; i.e., by considering all of the possible cases $\mathtt{C}_i$ —see `walk` above. In Agda, they are introduced with a `data` declaration; an intricate example below defines the datatype of lists of a particular length.

`Arabic` $\ell$ since the usual greeting "hello" (lit. "peace be upon you") is understandable by all Arabic speakers, whereas we may declare `ShakoMako : Arabic Iraq` since this question form "how are you" (lit. "what is your colour") is specific to the Iraqi Arabic dialect.

| Concept | Formal Name | Scenarios |
|---|---|---|
| “Two things” | Σ, `A × B`, records | One scenario with two payloads |
| “One from a union” | Sums `A + B`, unions | Two scenarios, each with one payload |
| “A sequence of things” | Lists, Vectors, ℕ | Empty and non-empty scenarios |
| “Truth values” | Booleans 𝔹 | Two scenarios with *no* payloads |
| “A pointer or reference” | `Maybe τ` | Two scenarios; successful or `null` |
| “Equality of two things” | Propositional _≡_ | One scenario; discussed later |

Many useful ideas arise as grammars

Vectors —ℕ-indexed Lists

```
data Vec {ℓ : Level} (A : Set ℓ) : ℕ → Set ℓ where
  []  : Vec A 0
  _::_ : {n : ℕ} → A → Vec A n → Vec A (1 + n)
```

Notice that, for a given type `A`, the type of `Vec A` is `ℕ → Set`. This means that `Vec A` is a family of types indexed by natural numbers: For each number `n`, we have a type `Vec A n`. One says `Vec` is *parameterised* by `A` (and ℓ), and *indexed* by `n`. They have different roles: `A` is the type of elements in the vectors, whereas `n` determines the ‘shape’ —length— of the vectors and so needs to be more ‘flexible’ than a parameter.

Notice that the indices say that the only way to make an element of `Vec A 0` is to use `[]` and the only way to make an element of `Vec A (1 + n)` is to use `_::_`. Whence, we can write the following safe function since `Vec A (1 + n)` denotes non-empty lists and so the pattern `[]` is impossible.

Safe Head

```
head : {A : Set} {n : ℕ} → Vec A (1 + n) → A
head (x :: xs) = x
```

The ℓ argument means the `Vec` type operator is *universe polymorphic*: We can make vectors of, say, numbers but also vectors of types. Levels are essentially natural numbers: We have `lzero` and `lsuc` for making them, and _⊔_ for taking the maximum of two levels. *There is no universe of all universes:* $\texttt{Set}_n$ has type $\texttt{Set}_{n+1}$ *for any n*, however the *type* `(n : Level) → Set n` is *not* itself typeable —i.e., is not in $\texttt{Set}_l$ for any `l`— and Agda errors saying it is a value of `Setω`.

Functions are defined by pattern matching, and must cover all possible cases. Moreover, they must be terminating and so recursive calls must be made on structurally smaller arguments; e.g., `xs` is a sub-term of `x :: xs` below and catenation is defined recursively on the first argument. Firstly, we declare a *precedence rule* so we may omit parenthesis in seemingly ambiguous expressions.

Catenation is a $++ \longrightarrow +$ Homomorphism

```
infixr 40 _++_

_++_ : {A : Set} {n m : ℕ} → Vec A n → Vec A m → Vec A (n + m)
[]       ++ ys  =  ys
(x :: xs) ++ ys  =  x :: (xs ++ ys)
```

Notice that the **type encodes a useful property**: The length of the catenation is the sum of the lengths of the arguments.

### 2.4.3 ADT Example: Propositional Equality

In this section, we present a notion of equality as an algebraic data type. Equality is a notoriously difficult concept, even posing it is non-trivial: "When are two things equal?" sounds absurd, since the question speaks about two things and two different things cannot be the same one thing. For us, equality is the smallest possible reflexive relation: Any relation $\mathcal{R}$ that relates things to themselves —such that $x\,\mathcal{R}\,x$ for any $x$— must necessarily contain the propositional equality relation; i.e., `_≡_` $\subseteq\ \mathcal{R}$.

For a type `A` and an element `x` of `A`, we define the family of types/proofs of "being equal to $x$" by declaring only one inhabitant at index `x`.

Propositional Equality

```
data _≡_ {A : Set} : A → A → Set
  where
    refl : {x : A} → x ≡ x
```

This states that `refl {x}` is a proof of `l ≡ r` whenever `l` and `r` simplify, by definition chasing only, to `x` —i.e., both `l` and `r` have `x` as their normal form.

This definition makes it easy to prove Leibniz's substitutivity rule, "equals for equals":

Transport along proofs

```
{- If l ≡ r and we have P l, then we also have P r too! -}
subst : {A : Set} {P : A → Set} {l r : A} → l ≡ r → P l → P r
subst refl it = it
```

Why does this work? An element of `l ≡ r` must be of the form `refl {x}` for some canonical form `x`; but if `l` and `r` are both `x`, then `P l` and `P r` are the *same type*. Pattern matching on a proof of `l ≡ r` gave us information about the rest of the program's type. By the same reasoning, we can prove that equality is the least reflexive relation.

Propositional equality is the least reflexive relagtion

```
-- If 𝓡 is reflexive, then it contains _≡_
lrr : ∀ {X} {_𝓡_ : X → X → Set}
    → (reflᵣ : ∀ {x} → x 𝓡 x)
    → ∀ {x y} → x ≡ y → x 𝓡 y
lrr reflᵣ refl = reflᵣ

-- If 𝓡 contains _≡_, then it is reflexive
lrr˘ : ∀ {X} {_𝓡_ : X → X → Set}
     → (𝓡-contains-≡ : ∀ {x y} → x ≡ y → x 𝓡 y)
     → ∀ {x} → x 𝓡 x
lrr˘ 𝓡-contains-≡ {x} = 𝓡-contains-≡ refl

-- “𝓡 is reflexive precively when it contains _≡_”
-- This follows from (lrr) and (lrr˘), and is sometimes
-- “the” definition of reflexivity.
```

One says $l \equiv r$ is *definitionally equal* when both sides are indistinguishable after all possible definitions in the terms $l$ and $r$ have been used. In contrast, the equality is «</propositionally equal/»> when one must perform actual work, such as using inductive reasoning. In general, if there are no variables in $l \equiv r$ then we have definitional equality —i.e., simplify as much as possible then compare— otherwise we have propositional equality —real work to do. Below is an example about the types of vectors.

Examples of Propositional and Definitional Equality

```
definitional : ∀ {A} → Vec A 5 ≡ Vec A (2 + 3)
definitional = refl

propositional : ∀ {A m n} → Vec A (m + n) ≡ Vec A (n + m)
propositional = {!!}
```

### 2.4.4 ADTs as 𝒲-types

Grammars, `data` declarations, *describe* the *smallest* language that has the constructors as words. What if no such language exists? Indeed, not all grammars are ‘sensible’ in that they define a language. For instance, `N` below is a language of only **one word**, `MakeN`; whereas `No` is a language with **no words**, since to form a phrase `MakeNo n` first requires we form `n`, which leads to infinite regress, and so there are no *finite* words. Even worse, `Noo` describes no language at all and Agda says that it is `not strictly postive`.

Describing Possibly Non-Existent Languages

```
data N : Set where
  MakeN : N

data No : Set where
  MakeNo : No → No

data Noo : Set where
 MakeNoo : (Noo → Noo) → Noo
```

How do we know if a grammar describes a language that *actually exists*? Suppose `T` is defined by $n$ constructors $\texttt{C}_i : \tau_i(\texttt{T}) \to \texttt{T}$, which may mention `T` in their payload $\tau_i(\texttt{T})$. Then we have a type operation $\mathbf{F}\ \texttt{X} = (\Sigma\ \texttt{i} : \texttt{Fin n} \bullet \tau_i(\texttt{X}))$, where `Fin n` is the type of natural numbers less than `n`. The type `T` describes a language `X` that *contains* all the constructors; i.e., "it can distinguish the constructors, along with their payloads"; i.e., there is a method $\mathbf{F}\ \texttt{X} \to \texttt{X}$ that shows how the descriptive constructors $\mathbf{F}\ \texttt{X}$ can be viewed as values of `X`. More concretely, the type `N` above has one constructor `MakeN` which takes an empty tuple of arguments, denoted $\mathbb{1} = \{\ ()\ \}$, and so it has $\mathbf{F}\ \texttt{X} \approx \mathbb{1}$ and so $(\mathbf{F}\ \texttt{X} \to \texttt{X}) \approx (\mathbb{1} \to \texttt{X}) \approx \texttt{X}$; whence any non-empty collection `X` is described by $\mathbf{F}$; but the **smallest** such language is a singleton language with one element that we call `MakeN`. **ADTs describe the smallest languages generated by their constructors**.

Important Observation

Recall that we earlier observed that $\Pi$ and $\Sigma$ could be thought of as way to interpret a contextual judgement; so too a judgement $\Gamma \vdash t : \tau$ could be interpreted as a term $t : \tau$ in the presence of the ADT described by some $\mathbf{F}$ which is obtained by treating all (or a select set of) names of $\Gamma$ as constructors.

Indeed, $\mathcal{W}$-types (introduced below) are essentially generalised signatures: $\mathcal{W}$ `A B` has `A` as 'function symbols' and each symbol `f : A` has 'type' `B f`. $\mathcal{W}$-types are not generalised signatures since they do not support optional definitions; which is a minor technicality: If $t$ has the associated definition `d`, then we may use " `let t = d in` $\mathcal{W}$ $\cdots$ " and repeated `let` clauses solve the issue of optional definitions.

The generic situation of 'containers' is described in [**DBLP:journals/jfp/AltenkirchGHMM15**].

Notice that we have again encountered the problem of a syntax that is "too powerful" for the concepts it denotes: We can declare grammars (ADTs) that do not describe *any* language. Since a grammar consists of a number of *disjoint* ("$\Sigma$") constructor clauses that take a *tuple* ("$\Pi$") of arguments, it suffices to consider when "polynomial"[11] descriptions

[11]Using exponential notation $Q^P = (P \to Q)$ along with subscript notation yields $\mathbf{F}\,X = \Sigma_{a:A} X^{B\,a}$, which is the shape of a polynomial. These notations and names are standard.

$\mathbf{F}$ `X = (Σ a : A • Π b : B a • X)` actually describe a language. That is, when is there a function $\mathbf{F}\, X \to X$ and what is the *smallest* $X$ with such a function? The values of $\mathbf{F}\, X$ are pairs $(a, f)$ where $a : A$ and $f : B\, a \to X$; so we may take the collection of *only* such pairs to be the language described by $\mathbf{F}$, and it is thus the smallest such collection. This[12] language is called a **$\mathcal{W}$-type**.

**Descriptions of Languages That Necessarily Exist**

`(𝒲 a : A • B a)` is the type of well-founded[a] trees with node "labels from $A$" and each node having "$B\, a$ many possible children trees". That is, it is the (inductive) language/type whose *constructors* are indexed by elements $a : A$, each with arity $B\, a$.

**$\mathcal{W}$-types in Agda**

```
-- The type of trees with B-branching degrees
data 𝒲 (A : Set) (B : A → Set) : Set where
  sup : (a : A) → (B a → 𝒲 A B) → 𝒲 A B
```

In particular, `𝒲` $i$ `: Fin n • B` $i$ is essentially the `data` declaration of `n` constructors where the $i$-th constructor takes arguments of 'shape' `B` $i$.
E.g., in Agda syntax, `ℕ ≅ 𝒲 (Fin 2) λ{zero → Fin 0; (suc zero) → Fin 1}`.

[a]See [**hottbook**] and [**emmenegger18:_w**] for an introduction to $\mathcal{W}$-types. The dual, non-well-founded (coinductive) types, are called *M-types* and they are derivable from W-types; see [**DBLP:journals/corr/AhrensCS15**]. The generic concept of 'containers' is described in [**DBLP:journals/jfp/AltenkirchGHMM15**].

To further understand $\mathcal{W}$-types, consider the type `Rose A` of "multi-branching trees with leaves from $A$". *$\mathcal{W}$-types generalise the idea of rose trees:* Each list of children trees `xs : List (Rose A)` can be equivalently[13] replaced by a *tabulation* `cs : Fin (length xs) → Rose A` that tells the $i$-th child of `xs`. That is, **$\mathcal{W}$-types are trees with branching degrees $(B\, a)_{a:A}$.**

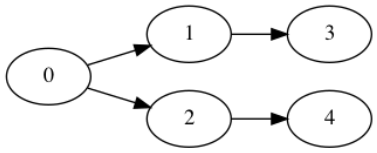
[12]Categorically speaking, polynomial functors —i.e., type formers of the shape $\mathbf{F}$ `X = Σ a : A • Π b : B a • X`, "sums of products" or a "disjoint union of possible constructors and their arguments"— have "initial algebras" named $\mathbf{W}$ `= (𝒲 a : A • B a)`, which are the smallest languages described by $\mathbf{F}$. That is, $\mathcal{W}$-types are the initial algebras of polynomial functors; that is, $\mathbf{F}$ has an initial algebra `sup` : $\mathbf{F}\ \mathbf{W} \to \mathbf{W}$. Moreover, every strictly positive type operator can be expressed in the same shape as $\mathbf{F}$ and so they all have an initial algebra —for details see [**Dybjer_1997**]. Inductive families arise as indexed $\mathcal{W}$-types which are initial algebras for dependent polynomial functors, and [**Gambino_2004**] have shown them to be constructible from non-dependent ones in locally cartesian closed categories. That is, indexed $\mathcal{W}$-types can be obtained from ordinary $\mathcal{W}$-types. See also [**DBLP:conf/icalp/AbbottAG04**].

[13]Since every functon `Fin n → X` can be 'tabulated' as a `List X` value of length `n` —i.e., `(Σ xs : List A • length xs ≡ n) ≅ (Fin n → A)`— we have that `Rose′ A ≅ Rose A`.

Rose trees

```
data Rose (A : Set) : Set where
  Node : (parent : A) (children : List (Rose A)) → Rose A

example : Rose ℕ
example = MkRose 0  (MkRose 1 (MkRose 3 [] :: [])
                   :: MkRose 2 (MkRose 4 [] :: []) :: [])
```

The `example` tree is shown diagrammatically below.



We can easily recast the `Rose` type and the example as a $\mathcal{W}$-type. In particular, notice that in the construction of `example′`, each node construction `sup (a, n) cs` indicates that the label is `n` and the number of children the node has is `n`. That is, the choice of using lists or vectors in the design of `Rose` is forced to being (implicitly and essentially) vectors in the construction of `Rose′`.

Rose trees

```
Rose′ : Set → Set
Rose′ A = 𝒲 (A × ℕ) λ{ (a , ♯children)  → Fin ♯children }

example′ : Rose′ ℕ
example′ = sup ((0 , 2))
             λ { zero       → sup (1 , 1) λ {zero → sup (3 , 0) λ ()}
               ; (suc zero) → sup (2 , 1) λ {zero → sup (4 , 0) λ ()}}
```

Similar to rose trees, `𝒲 a : Fin n • Fin 0` is an enumerated type having `n` constants, such as the Booleans. That is, if `B a` is empty for all `a`, then trees in `𝒲 a : A • B a` have no subtrees, and hence have 'height' 0.

The *height* of a tree, is an ordinal, and is defined to be the supremum[14] —i.e., the least upper bound— of the height of its elements. This may be reason why the only constructor of $\mathcal{W}$-types is named `sup`.

$$\mathsf{height}\,(\sup\ a\,\mathsf{child}) \;=\; \sup_{i:B\,a}\,(\mathsf{height}\,(\mathsf{child}\ i) + 1)$$

[14]The supremum of the empty set is, by definition, 0.

$$\sup \emptyset = 0$$

Hence, if any (child) tree is empty, then its height is 0.

In contrast, `𝒲 a : A • Fin n` is a data type with `A`-many clauses that *each* make `n` recursive calls; this is an *empty type* since every construction requires `n` many existing constructions —however, it is still a type, unlike `Noo` above. That is[15], if `B a` is non-empty for all `a`, then `𝒲 a : A • B a` is empty, since in order to form an element `sup a c`, we need to have defined before-hand `c(b) : (𝒲 a : A • B a)` for each one of the elements `b` of `B a`.

Unlike generalised signatures which do not possess a singular semantics, Agda `data` declarations are pleasant way to write 𝒲-types.

### 2.4.5 Modules —Namespace Management; ΠΣ-types

For now, Agda modules are not first-class[16] constructs and essentially only serve to delimit namespaces, thereby avoiding name clashes. They use is exemplified by the following snippets.

A Simple Module

```
module A where

  𝒩 : Set
  𝒩 = ℕ

  private
    x : ℕ
    x = 3

  y : 𝒩
  y = x + 1
```

Using It

```
use₀ : A.𝒩
use₀ = A.y

use₁ : ℕ
use₁ = y
  where open A

open A

use₂ : ℕ
use₂ = y
```

Parameterised Modules

```
module B
    (x : ℕ)
  where
    y : ℕ
    y = x + 1
```

Name = Function

```
exposed
    : (x : ℕ)
    → ℕ
exposed = B.y
```

Using Them

```
use′₀ : ℕ
use′₀ = B.y 3

module C = B 3

use : ℕ
use = C.y

use′₁ : ℕ
use′₁ = y
  where
    open B 3
```

When opening a module, we can control which names are brought into scope with the `using`, `hiding`, and `renaming` keywords.

All names in a module are public, unless declared `private`. Public names may be accessed by qualification or by opening them locally or globally. Modules may be parameterised by arbitrarily many values and types —but not by other modules.

[15] A 𝒲-type is empty precisely when it has no nullary constructor; See exercise 5.17 of [**hottbook**].

$$\neg(\mathcal{W}\ a\ :\ A\ \bullet\ B\ a) \cong \neg\ (\Sigma\ a\ :\ A\ \bullet\ \neg\ B\ a)$$

[16] A *first-class citizen* is a citizen that is not treated differently by having their rights reduced. In particular, first-class citizens may be serviced ('treated as data') by other citizens; *second-class citizens* can only provide a service and do not themselves have the right to be serviced.

| | |
|---|---|
| `open M hiding (`$n_0$`; ...; `$n_k$`)` | Essentially treat $n_i$ as private |
| `open M using (`$n_0$`; ...; `$n_k$`)` | Essentially treat *only* $n_i$ as public |
| `open M renaming (`$n_0$` to `$m_0$`; ...; `$n_k$` to `$m_k$`)` | Use names $m_i$ instead of $n_i$ |

Module combinators supported in the current implementation of Agda

Modules are essentially implemented as syntactic sugar: Their declarations are treated as top-level functions that take the parameters of the module as extra arguments. In particular, it may appear that module arguments are ‘shared’ among their declarations, but this is not so —see the `exposed` function above.

Parameterised Agda modules are generalised signatures that have all their parameters first then followed by only by named symbols that must have term definitions. Unlike generalised signatures which do not possess a singular semantics, Agda modules are pleasant way to write ΠΣ-types —the parameters are captured by a Π type and the defined named are captured by Σ-types as in “ Π `parameters` • Σ `body` ”.

### 2.4.6 Records — Σ-types

An Agda record type is *presented* like a generalised signature, except parameters may either appear immediately after the record’s name declaration or may be declared with the `field` keyword; other named symbols must have an accompanying term definition. Unlike generalised signatures which do not possess a singular semantics, Agda records are essentially a pleasant way to write Σ-types. The nature of records is summarised by the following equation.

`record` ≈ `module` + `data` with one constructor

The class of types along with a value picked out

```
record PointedSet : Set₁ where
  constructor MkIt  -- Optional
  field
    Carrier : Set
    point   : Carrier

  -- It's like a module,
  -- we can add definitions
  blind : {A : Set}
        → A → Carrier
  blind = λ a → point
```

Defining Instances

```
ex₀ : PointedSet
ex₀ = record { Carrier = ℕ
             ; point   = 3 }

ex₁ : PointedSet
ex₁ = MkIt ℕ 3

open PointedSet

ex₂ : PointedSet
Carrier ex₂ = ℕ
point   ex₂ = 3
```

Two tuples are the same when they have the same components, likewise a record is (extensionaly) defined by its projections, whence *co-patterns*: The declarations
`r = record {f`$_i$` = d`$_i$`}` and `f`$_i$` r = d`$_i$, for field names `f`$_i$, are the same; they define values of record types. See `ex`$_2$ above for such an example.

To allow projection of the fields from a record, each record type comes with a module of the same name. This module is parameterised by an element of the record type and contains projection functions for the fields.

Simple Uses

```
use⁰ : ℕ
use⁰ = PointedSet.point ex₀

use¹ : ℕ
use¹ = point
    where open PointedSet ex₀

open PointedSet

use² : ℕ
use² = blind ex₀ true
```

Pattern Matching on Records

```
use³ use⁴ : (P : PointedSet)
          → Carrier P

use³ record {Carrier = C
            ; point = x}
  = x

use⁴ (MkIt C x)
  = x
```

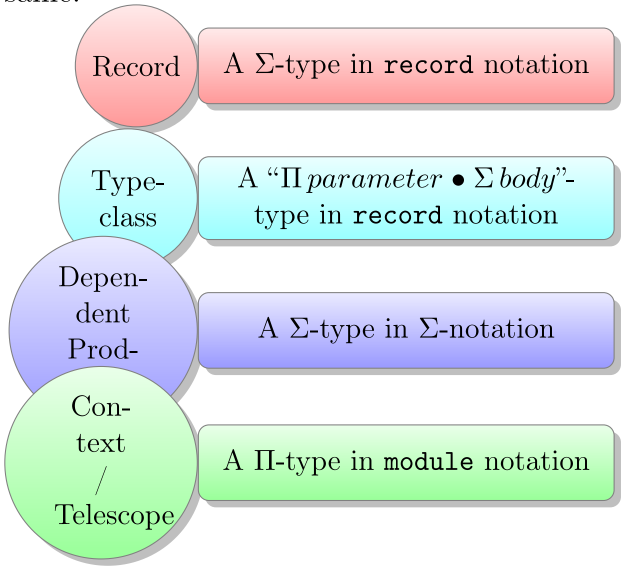
Records are `data` declarations whose one and only constructor is named
`record {f`$_i$` = _}`, where the `f`$_i$ are the filed names; above we provided `MkIt` as an optional alias. As such, above we could pattern match on records using either constructor name.

So much for records.

## 2.5 Facets of Structuring Mechanisms

In this section we provide a demonstration that with dependent-types we can show records, direct dependent types, and contexts —which in Agda may be thought of as parameters to a module— are interdefinable. Consequently, we observe that the structuring mechanisms provided by the current implementation of Agda —and other DTLs— have no real differences aside from those imposed by the language and how they are generally utilised. More importantly, this demonstration indicates our proposed direction of identifying notions of packages is on the right track.

Our example will be implementing a monoidal interface in each format, then presenting *views* between each format and that of the `record` format. Furthermore, we shall also construe each as a typeclass, thereby demonstrating that typeclasses are, essentially, not only a selected record but also a selected *value* of a dependent type —incidentally this follows from the previous claim that records and direct dependent types are essentially the same.



Record
A Σ-type in record notation
Type-class
A "Π *parameter* • Σ *body*"-type in record notation
Dependent Prod-
A Σ-type in Σ-notation
Context / Telescope
A Π-type in module notation

### 2.5.1 Three Ways to Define Monoids

A **monoid** is a collection, say `Carrier`, along with an operation, say `_⨾_`, on it and a chosen point, say `Id`, from that collection. **Monoids model composition:** We have a bunch of things called `Carrier` —such as programs or words—, we have a way to 'mix' or 'compose' two things `x` and `y` to get a third `x ⨾ y` —such as forming a big program from smaller pieces or a story from words— which has an selected 'empty' thing that does not affect composition —such as the do-nothing program or the 'empty word' which does not add content to a story. The type of monoids is formalised below as `Monoid-Record`; additionally, we have the derived result: `Id`-entity can be popped-in and out as desired.

Monoids as Agda Records

```
record Monoid-Record : Set₁ where
  infixl 5 _⨾_
  field
    -- Interface
    Carrier  : Set
    Id       : Carrier
    _⨾_      : Carrier → Carrier → Carrier

    -- Constraints
    lid   : ∀{x}     → (Id ⨾ x) ≡ x
    rid   : ∀{x}     → (x ⨾ Id) ≡ x
    assoc : ∀ x y z → (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)

  -- derived result
  pop-Id-Rec : ∀ x y  →  x ⨾ Id ⨾ y  ≡  x ⨾ y
  pop-Id-Rec x y = cong (_⨾ y) rid

open Monoid-Record {{...}} using (pop-Id-Rec)
```

Instance Resolution

The double curly-braces `{{...}}` serve to indicate that the given argument is to be found by *instance resolution*. For example, if we declare `it : {{e : A}} → B`, then `it` is a `B` value that is formed using an `A` value; but which `A` value? Unlike a function which requires the `A` value as input, `it` will "look up" an `A` value in the list of names that are marked for look-up by the keyword `instance`. If multiple `A` values are marked for look-up, it is not clear which one should be used; as such, *at most one*[a] value can be provided for lookup and this value is called "the declared `A`-instance", whence the name 'instance resolution'. Recall that Agda records automatically come with an associated module, and so the `open` clause, above, makes the name `pop-Id-Rec : {{M : Monoid-Record}} → (x y : Monoid-Record.Carrier M) → ...` accessible; in-particular, this name uses instance resolution: The derived result, `pop-Id-Rec`, can be invoked without having to mention a monoid, provided a unique `Monoid-Record` value is declared for instance search —otherwise one must use named instances **named_instances**. We will return to actually declaring and using instances in the next section.

[a]More accurately, there needs to be *a unique instance that solves local constraints.* Continuing with `it`, any call to `it` will occur in a context $\Gamma$ that will include inferred types and so when an `A`-valued is looked-up it suffices to find a *unique* value `e` such that $\Gamma \vdash$ `e : A`. More concretely, suppose `A = ℕ × ℕ`, `B = ℕ≡`, and `it {{(x , y)}} = x` and we declared two ℕumbers for instance search, `p = (0 , 10)` and `q = (1, 14)`. Then in the call site `go : it ≡ 1; go = refl`, the use of `refl` means both sides of the equality must be identical and so `it {{e}}` must have the `e` chosen to make the equality true, but only `q` does so and so it is chosen. However, if instead we had defined `p = (1 , 10)`, then both `p` and `q` could be used and so there is no local solution; prompting Agda to produce an error.

A value of `Monoid-Record` is essentially a tuple `record{Carrier = C; ...}`; so the carrier is *bundled at the value level.* If we to speak of "monoids with the specific carrier $\mathcal{X}$", we need to *bundle the carrier at the type level.* This is akin to finding the carrier "dynamically, at runtime" versus finding it "statically, at typechecking time".

Monoids as Typeclasses

```
record MonoidOn (Carrier : Set) : Set₁ where
  infixl 5 _⨾_
  field
    Id    : Carrier
    _⨾_   : Carrier → Carrier → Carrier
    lid   : ∀{x} → (Id ⨾ x) ≡ x
    rid   : ∀{x} → (x ⨾ Id) ≡ x
    assoc : ∀ x y z → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)

  pop-Id-Tc : ∀ x y →  x ⨾ Id ⨾ y  ≡  x ⨾ y
  pop-Id-Tc x y = cong (_⨾ y) rid

open MonoidOn {{...}} using (pop-Id-Tc)
```

Alternatively, in a DTL we may encode the monoidal interface using dependent products **directly** rather than use the syntactic sugar of records. Recall that `Σ a : A • B a` denotes the type of pairs `(a , b)` where `a : A` and `b : B a` —i.e., a record consisting of two fields— and it may be thought of as a constructive analogue to the classical set comprehension `{x : A | B x}`.

Monoids as Dependent Sums

```
-- Type alias
Monoid-Σ  :  Set₁
Monoid-Σ  =    Σ Carrier : Set
             • Σ Id : Carrier
             • Σ _⨾_ : (Carrier → Carrier → Carrier)
             • Σ lid : (∀{x} → Id ⨾ x ≡ x)
             • Σ rid : (∀{x} → x ⨾ Id ≡ x)
             • (∀ x y z → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z))

pop-Id-Σ : ∀ {{M : Monoid-Σ}}
             (let Id  = proj₁ (proj₂ M))
             (let _⨾_ = proj₁ (proj₂ (proj₂ M)))
           → ∀ (x y : proj₁ M) →  (x ⨾ Id) ⨾ y  ≡  x ⨾ y
pop-Id-Σ {{M}} x y = cong (_⨾ y) (rid {x})
           where  _⨾_   = proj₁ (proj₂ (proj₂ M))
                  rid   = proj₁ (proj₂ (proj₂ (proj₂ (proj₂ M))))
```

Observe the lack of informational difference between the presentations, yet there is a *Utility Difference: Records give us the power to name our projections <u>directly</u> with possibly* *meaningful names.* Of course this could be achieved indirectly by declaring extra functions; e.g.,

```
Agda
Carrierₜ : Monoid-Σ → Set
Carrierₜ = proj₁
```

We will refrain from creating such boiler plate —that is, *records allow us to omit such mechanical boilerplate.*

Of the renditions thus far, the Σ rendering makes it clear that a monoid could have any subpart as a record with the rest being dependent upon said record. For example, if we had a semigroup[17] type, we could have declared a monoid to be a semigroup with additional pieces:

```
Monoid-Σ  =  Σ S : Semigroup • Σ Id : Semigroup.Carrier S • ⋯
```

There are a large number of hyper-graphs indicating how monoidal interfaces could be built from their parts, we have only presented a stratified view for brevity. In particular, `Monoid-Σ` is the extreme unbundled version, whereas `Monoid-Record` is the other extreme, and there is a large spectrum in between —all of which are somehow isomorphic[a]; e.g., `Monoid-Record ≅ Σ C : Set • MonoidOn C`. Our envisioned system would be able to derive any such view at will **casl_overview** and so programs may be written according to one view, but easily repurposed for other view with little human intervention.

[a]For this reason —namely that records are existential closures of a typeclasses— typeclasses are also known as "constraints, or predicates, on types".

## 2.5.2 Instances and Their Use

Instances of the monoid types are declared by providing implementations for the necessary fields. Moreover, as mentioned earlier, to support instance search, we place the declarations in an `instance` clause.

[17]A *semigroup* is like a monoid except it does not have the `Id` element.

Instance Declarations

```
instance
   ℕ-Rec : Monoid-Record
   ℕ-Rec = record { Carrier = ℕ ; Id = 0 ; _⨾_ = _+_
                  ; lid =  +-identityˡ _  ; rid = +-identityʳ _
                  ; assoc = +-assoc }

   ℕ-Tc : MonoidOn ℕ
   ℕ-Tc = record { Id = 0; _⨾_ = _+_ ; lid = +-identityˡ _
                 ; rid = +-identityʳ _ ; assoc = +-assoc }

   ℕ-Σ : Monoid-Σ
   ℕ-Σ = ℕ , 0 , _+_ , +-identityˡ _ , +-identityʳ _ , +-assoc
```

Interestingly, notice that the grouping in ℕ-Σ is just an unlabelled (dependent) product, and so when it is used below in `pop-Id-Σ` we project to the desired components. Whereas in the `Monoid-Record` case we could have projected the carrier by `Carrier M`, now we would write `proj₁ M`.

No Monoids Mentioned at Use Sites

```
ℕ-pop-0-Rec ℕ-pop-0-Tc ℕ-pop-0-Σ : (x y : ℕ) → x + 0 + y ≡  x + y

ℕ-pop-0-Rec  = pop-Id-Rec
ℕ-pop-0-Tc   = pop-Id-Tc
ℕ-pop-0-Σ     = pop-Id-Σ
```

With a change in perspective, we could treat the `pop-0` implementations as a form of *polymorphism*: The result is independent of the particular packaging mechanism; record, typeclass, Σ, it does not matter.

Finally, since we have already discussed the relationship between `Monoid-Record` and `MonoidOn`, let us exhibit views between the Σ form and the `record` form.

Monoid-Record and Monoid-Σ *represent* the same data

```
{- Essentially moved from record{⋯} to product listing -}
from : Monoid-Record → Monoid-Σ
from M  =  let open Monoid-Record M
           in Carrier , Id , _⨾_ , lid , rid , assoc

from-record-to-usual-type M  =  Carrier , Id , _⨾_ , lid , rid , assoc

{- Organise a tuple componenets as implementing named fields -}
to : Monoid-Σ → Monoid-Record
to (c , id , op , lid , rid , assoc)  = record { Carrier = c
                                               ; Id      = id
                                               ; _⨾_     = op
                                               ; lid     = lid
                                               ; rid     = rid
                                               ; assoc   = assoc
                                               }
```

Furthermore, by definition chasing, `refl`-exivity, these operations are seen to be inverse of each other. Hence we have two faithful non-lossy protocols for reshaping our grouped data.

## 2.5.3 A Fourth Definition —Contexts

In our final presentation, we construe the grouping of the monoidal interface as a sequence of *variable* : *type* declarations —i.e., a Context or 'telescope'. Since these are not top level items by themselves, in Agda, we take a purely syntactic route by positioning them in a `module` declaration as follows.

Monoids as Telescopes

```
module Monoid-Telescope-User
  (Carrier : Set)
  (Id      : Carrier)
  (_⨾_     : Carrier → Carrier → Carrier)
  (lid     : ∀{x} → Id ⨾ x ≡ x)
  (rid     : ∀{x} → x ⨾ Id ≡ x)
  (assoc   : ∀ x y z → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z))
  where

  pop-Id-Tel : ∀(x y : Carrier)  →  (x ⨾ Id) ⨾ y  ≡  x ⨾ y
  pop-Id-Tel x y = cong (_⨾ y) (rid {x})
```

Squint and They're The Same

Notice that this is nothing more than the named fields of `Monoid-Record` but not[a] bundled. Additionally, if we insert a Σ before each name we essentially regain the `Monoid-Σ` formulation. It seems contexts, at least superficially, are a nice middle ground between the previous two formulations. For instance, if we *syntactically*, visually, move the
`Carrier : Set` declaration one line above, the resulting setup looks eerily similar to the typeclass formulation of records.

[a]Records let us put things in a bag and run around with them, whereas telescopes amount to us running around with all of our things in our hands —hoping we don't drop (forget) any of them.

As promised earlier, we can regard the above telescope as a record:

Agda

```
{- No more running around with things in our hands. -}
{- Place the telescope parameters into a nice bag to hold on to. -}
record-from-telescope : Monoid-Record
record-from-telescope
  = record { Carrier = Carrier
           ; Id      = Id
           ; _⨾_     = _⨾_
           ; lid     = lid
           ; rid     = rid
           ; assoc   = assoc
           }
```

The structuring mechanism `module` is not a first class citizen in Agda. As such, to obtain the converse view, we work in a parameterised module.

Agda

```
module record-to-telescope (M : Monoid-Record) where

  -- Treat record type as if it were a parameterised module type,
  -- instantiated with M.
   open Monoid-Record M

  -- Actually using M as a telescope
  open Monoid-Telescope-User Carrier Id _⨾_ lid rid assoc
```

Notice that we just listed the components out —rather reminiscent of the formulation `Monoid-Σ`. This observation only increases confidence in our thesis that there is no real distinctions of packaging mechanisms in DTLs. Similarity, instantiating the telescope approach to a natural number monoid is nothing more than listing the required components.

Agda

```
open Monoid-Telescope-User ℕ 0 _+_ (+-identityˡ _) (+-identityʳ _) +-assoc
```

This instantiation is nearly the same as the definition of ℕ-Σ; with the primary syntactical difference being that this form had its arguments separated by spaces rather than commas!

Agda

```
ℕ-pop-Tel  : ∀(x y : ℕ)  →  x + 0 + y  ≡  x + y
ℕ-pop-Tel  =   pop-Id-Tel
```

It is interesting to note that this presentation is akin to that of `class`-es in C#/Java languages: The interface is declared in one place, monolithic-ly, as well as all derived operations there; if we want additional operations, we create another module that takes that given module as an argument in the same way we create a class that inherits from that given class.

Demonstrating the interdefinablity of different notions of packaging cements our thesis that it is essentially *utility* that distinguishes packages more than anything else —just as `data` language's words (constructors) have their meanings determined by *utility*. Consequently, explicit distinctions have lead to a duplication of work where the same structure is formalised using different notions of packaging. In chapter 4 we will show how to avoid duplication by coding against a particular 'package former' rather than a particular variation thereof —this is akin to a type former.

## 2.6 Contexts are Promising

The current implementation of the Agda language **agda_overview**; **agda_thesis** has a notion of second-class modules which may contain sub-modules along with declarations and definitions of first-class citizens. The intimate relationship between records and modules is perhaps best exemplified here since the current implementation provides a declaration to construe a record as if it were a module —as demonstrated in the previous section. This observation is not specific to Agda, which is herein only used as a presentation language. Indeed, other DTLs (dependently-typed languages) reassure our hypothesis; the existence of a unified notion of package:

- ◇ **The centrality of contexts**

  The **Beluga** language has the distinctive feature of direct support for first-class contexts **beluga**. A term `t(x)` may have free variables and so whether it is well-formed, or what its type could be, depends on the types of its free variables, necessitating one to either declare them before hand or to write, in Beluga,

`[ x : T |- t(x) ]` for example. As argued in the previous section, contexts are essentially dependent sums. In contrast to Beluga, **Isabelle** is a full-featured language and logical framework that also provides support for named contexts in the form of 'locales' **locales**; **isabelle_locales**; unfortunately it is not a dependently-typed language.

◇ **Signatures as an underlying formalism**

**Twelf twelf_site** is a logic programming language implementing Edinburgh's Logical Framework **lf_meta_mechanisation**; **lf_has_isabelle**; **lf_fast_proof_checking** and has been used to prove safety properties of 'real languages' such as SML. A notable practical module system **lf_practical_modules** for Twelf has been implemented using signatures and signature morphisms.

◇ **Packages (modules) have their own useful language**

The current implementation of **Coq coq_implementation**; **coq_cat_experiences** provides a "copy and paste" operation for modules using the `include` keyword. Consequently it provides a number of module combinators, such as `<+` which is the infix form of module inclusion **coq_manual**. Since Coq module types are essentially contexts, the module type `X <+ Y <+ Z` is really the catenation of contexts, where later items may depend on former items. The **Maude maude**; **maude_module_algebra** framework contains a similar yet more comprehensive algebra of modules and how they work with Maude theories.

It is important to consider other languages so as to how see their communities treat module systems and what uses cases they are interested in. In the next section, we shall see a glimpse of how the Coq community works with packages, and, to make the discussion accessible, we shall provide Agda translations of Coq code.

## 2.7 Coq Modules as Generalised Signatures

Module Systems parameterise programs, proofs, and tactics over structures. In this section, we shall form a library of simple graphs[18] to showcase how Coq's approach to packages is essentially in the same spirit[19] as the proposed definition of generalised signatures: A

[18]A **graph** models "lines and dots on a page"; i.e., it is a tuple `(V, E, tgt, src)` where sets `V` and `E` denote the dots ('vertices') and lines ('edges'), respectively, and the functions `src, tgt : E → V` assign a 'source' and a 'target' dot (vertex) to each line (edge); so we do not have any "dangling lines": All lines on the page must be between drawn dots. In a simple graph, every edge is determine by its source and target points, so we can instead present a graph as a *set* `V` and a **dependent-type** `E : V × V → Type` where `E x y` denotes the collection of edges starting at `x` and ending at `y`. The code fragments of this section use the second form, for brevity.

[19]With this observation, it is only natural to wonder why Coq is not used as the presentation language in-place of Agda. We could rationalise our choice with technical attacks against Coq —e.g., tactics are evil since they render the concept of 'proof' as secondary— but they would not reflect reality: Coq is a delight to use, but Agda's community-adopted Unicode support and our own experiences with it biased our choice.

sequence of name-type-definition tuples where the definition may be omitted. To make the Coq accessible to readers, we will provide an Agda translation that only uses the `record` construct in Agda —completely ignoring the `data` and `module` forms which would otherwise be more natural in certain scenarios below— in order to demonstrate that *all packaging concepts essentially coincide in a DTL.*

Along the way, we refer to aspects of Agda that we found convenient and desirable that we chose it as a presentation language instead Coq and other equally appropriate DTLs.

In Coq, a `Module Type` contains the signature of the abstract structure to work from; it lists the `Parameter` and `Axiom` values we want to use, possibly along with notation declaration to make the syntax easier.

Graphs —Coq

```
Module Type Graph.
  Parameter Vertex : Type.
  Parameter Edges : Vertex -> Vertex -> Prop.

  Infix "<=" := Edges : order_scope.
  Open Scope order_scope.

  Axiom loops : forall e, e <= e.
  Parameter decidable : forall x y, {x <= y} + {not (x <= y)}.
  Parameter connected : forall x y, {x <= y} + {y <= x}.
End Graph.
```

Graphs —Agda

```
record Graph : Set₁ where
  field
    Vertex    : Set
    _⟶_        : Vertex → Vertex → Set
    loops     : ∀ {e} → e ⟶ e
    decidable : ∀ x y → Dec (x ⟶ y)
    connected : ∀ x y → (x ⟶ y) ⊎ (y ⟶ x)
```

Notice that due to Agda's support for mixfix Unicode lexemes, we are able to use the evocative arrow notation `_⟶_` for edges directly. In contrast, Coq uses ASCII order notation *after* the type of edges is declared. In contrast to Agda, conventional Coq distinguishes between value parameters and proofs, thereby using the keywords `Parameter` and `Axiom` to, essentially, accomplish the same thing.

In Coq, to form an instance of the graph module type, we define a module that satisfies the module type signature. The `_<:_` declaration requires us to have definitions and theorems with the same names and types as those listed in the module type's signature. In contrast, the Agda form below explicitly ties the signature's named fields with their implementations, rather than inferring it.

**Birds' Eye View**

The following two snippets only serve to produce instances of graphs that can be used in subsequent snippets, as such their details are mostly irrelevant. They are present here for the sake of completeness and we rely on the reader to accept them for their overarching purpose —namely, to demonstrate how Coq's `Module Type`'s are close in spirit to the previously discussed notion of generalised signatures. For the curious reader, the next Coq snippet is annotated with comments explaining the tactics.

**Booleans are Graphs —Coq**

```
Module BoolGraph <: Graph.
  Definition Vertex := bool.
  Definition Edges  := fun x => fun y => leb x y.

  Infix "<=" := Edges : order_scope.
  Open Scope order_scope.

  Theorem loops: forall x : Vertex, x <= x.
    Proof.
    intros; unfold Edges, leb; destruct x; tauto.
    Qed.

  Theorem decidable: forall x y, {Edges x y} + {not (Edges x y)}.
    Proof.
      intros; unfold Edges, leb; destruct x, y.
      all: (right; discriminate) || (left; trivial).
  Qed.

  Theorem connected: forall x y, {Edges x y} + {Edges y x}.
    Proof.
      intros; unfold Edges, leb. destruct x, y.
      all: (right; trivial; fail) || left; trivial.
  Qed.
End BoolGraph.
```

**Booleans are Graphs —Agda**

```
BoolGraph : Graph
BoolGraph = record
                { Vertex = Bool
                ; _⟶_ = leb
                ; loops = b≤b
                -- I only did the case analysis, the rest was
               ↪  “auto”.
                ; decidable = λ{ true  true  → yes b≤b
                               ; true  false → no (λ ())
                               ; false true  → yes f≤t
                               ; false false → yes b≤b }
                -- I only did the case analysis, the rest was
               ↪  “auto”.
                ; connected = λ{ true true   → inj₁ b≤b
                               ; true false  → inj₂ f≤t
                               ; false true  → inj₁ f≤t
                               ; false false → inj₁ b≤b }
                }
```

We are now in a position to write a "module functor": A module that takes some `Module Type` parameters and results in a module that is inferred from the definitions and parameters in the new module; i.e., a parameterised module. E.g., here is a module that defines a minimum function.

Minimisation as a function on modules —Coq

```
Module Min (G : Graph).
  Import G. (* I.e., open it so we can use names in unquantifed form. *)
  Definition min a b : Vertex := if (decidable a b) then a else b.
  Theorem case_analysis: forall P : Vertex -> Type, forall x y,
        (x <= y -> P x) -> (y <= x -> P y) -> P (min x y).
  Proof.
    intros. (* P, x, y, and hypothesises H₀, H₁ now in scope*)
    (* Goal: P (min x y) *)
    unfold min. (* Rewrite “min” according to its definition. *)
    (* Goal: P (if decidable x y then x else y) *)
    destruct (decidable x y). (* Case on the result of decidable *)
    (* Subgoal 1: P x  ---along with new hypothesis H₃ : x ≤ y *)
    tauto. (* i.e., modus ponens using H₁ and H₃ *)
    (* Subgoal 2: P y  ---along with new hypothesis H₃ : ¬ x ≤ y *)
    destruct (connected x y).
    (* Subgoal 2.1: P y ---along with new hypothesis H₄ : x ≤ y *)
    absurd (x <= y); assumption.
    (* Subgoal 2.2: P y ---along with new hypothesis H₄ : y ≤ x *)
    tauto. (* i.e., modus ponens using H₂ and H₄ *)
  Qed.
End Min.
```

`Min` is a function-on-modules; the input type is a `Graph` value and the output module's type is inferred to be:

```
Sig Definition min : ⋯. Parameter case_analysis: ⋯. End
```

In contrast, Agda has no notion of signature, and so the declaration below only serves as a *namespacing* mechanism that has a parameter over-which new programs and proofs are abstracted —the primary purpose of module systems mentioned earlier.

Minimisation as a function on modules —Agda

```
record Min (G : Graph) : Set where
  open Graph G

  min : Vertex → Vertex → Vertex
  min x y with decidable x y
  ...| yes _  = x
  ...| no  _  = y

  case-analysis : ∀ {P : Vertex → Set} {x y}
                → (x ⟶ y  →  P x)
                → (y ⟶ x  →  P y)
                → P (min x y)
  case-analysis {P} {x} {y} H₀ H₁ with decidable x y | connected x y
  ... | yes x⟶y | _          = H₀ x⟶y
  ... | no ¬x⟶y | inj₁ x⟶y = ⊥-elim (¬x⟶y x⟶y)
  ... | no ¬x⟶y | inj₂ y⟶x = H₁ y⟶x

open Min
```

Let's apply the so called module functor. The `min` function, as shown in the comment below, now specialises to the carrier of the Boolean graph.

Applying module-to-module functions (part I) —Coq

```
Module Conjunction := Min BoolGraph.
Export Conjunction.
Print min.
(*
min =
fun a b : BoolGraph.Vertex => if BoolGraph.decidable a b then a else b
     : BoolGraph.Vertex -> BoolGraph.Vertex -> BoolGraph.Vertex
 *)
```

In the Agda setting, we can prove the aforementioned observation: The module is for namespacing *only* and so it has no non-trivial implementations.

Applying module-to-module functions (part I) —Agda

```
Conjunction = Min BoolGraph

uep : ∀ (p q : Conjunction) → p ≡ q
uep record {} record {} = refl

{- "min I" is the specialisation of "min" to the Boolean graph -}
_ : Bool → Bool → Bool
_ = min I where I : Conjunction; I = record {}
```

Unlike the previous functor, which had its return type inferred, we may explicitly declare a return type. E.g., the following functor is a `Graph → Graph` function.

A module-to-module function —Coq

```
Module Dual (G : Graph) <: Graph.
  Definition Vertex := G.Vertex.
  Definition Edges  x y : Prop := G.Edges y x.
  Definition loops := G.loops.
  Infix "<=" := Edges : order_scope.
  Open Scope order_scope.
  Theorem decidable: forall x y, {x <= y} + {not (x <= y)}.
    Proof.
      unfold Edges. pose (H := G.decidable). auto.
  Qed.
  Theorem connected: forall x y, {Edges x y} + {Edges y x}.
    Proof.
      unfold Edges.  pose (H := G.connected). auto.
  Qed.
End Dual.
```

Agda makes it clearer that this is a module-to-module function.

A module-to-module function —Agda

```
Dual : Graph → Graph
Dual G = let open Graph G in record
           { Vertex    = Vertex
           ; _⟶_        = λ x y →  y ⟶ x
           ; loops     = loops
           ; decidable = λ x y → decidable y x
           ; connected = λ x y → connected y x
           }
```

An example use would be renaming "min ↦ max" —e.g., to obtain meets from joins.

Applying module-to-module functions (part II) —Coq

```
Module Max (G : Graph).
  (* Module applications cannot be chained;
     intermediate modules must be named. *)
  Module DualG   := Dual G.
  Module Flipped := Min DualG.
  Import G.
  Definition max := Flipped.min.
  Definition max_case_analysis:
        forall P : Vertex -> Type, forall x y,
        (y <= x -> P x) -> (x <= y -> P y) -> P (max x y)
        := Flipped.case_analysis.
End Max.
```

Applying module-to-module functions (part II) —Agda

```
record Max (G : Graph) : Set where
  open Graph G
  private
    Flipped = Min (Dual G)
    I : Flipped
    I = record {}

  max : Vertex → Vertex → Vertex
  max = min I

  max-case-analysis : ∀ {P : Vertex → Set} {x y}
                 → (y ⟶ x  →  P x)
                 → (x ⟶ y  →  P y)
                 → P (max x y)
  max-case-analysis = case-analysis I
```

Here is a table summarising the two languages' features, along with JavaScript as a position of reference.

| | Signature | Structure |
|---|---|---|
| Coq | ≈ module type | ≈ module |
| Agda | ≈ record type | ≈ record value |
| JavaScript | ≈ prototype | ≈ JSON object |

Signatures and structures in Coq, Agda, and JavaScript

It is perhaps seen most easily in the last entry in the table, that modules and modules types are essentially the same thing: They are just partially defined record types. Again there is a **difference in the usage intent**:

| Concept | Intent |
|---|---|
| Module types | Any name may be opaque, undefined. |
| Modules | All names must be fully defined. |

Modules and module types only differ in intended utility

## 2.8 Problem Statement, Objectives, and Methodology

This section provides a statement of the problem that is addressed in this thesis. It also outlines the objectives of this thesis and discusses the methodology used to achieve those objectives.

### 2.8.1 Problem Statement

Currently, first-class module systems for dependently-typed languages are poorly *supported*. Modules $\mathcal{X}$ consisting of functions symbols, properties, and derived results are currently presented in the form `Is`$\mathcal{X}$: A module parameterised by function symbols and exposing derived results possibly with further, uninstantiated, proof obligations —that is, it is of the shape $\Pi^k\Sigma$, below, having parameters $p_i$ at the type level and fields $p_{w+i}$ at the body level.

$$\Pi^w\Sigma \;=\; \Pi\, p_1 : \tau_1 \bullet \Pi\, p_2 : \tau_2 \bullet \cdots \bullet \Pi\, p_w : \tau_w \bullet \Sigma\, p_{w+1} : \tau'_{w+1} \bullet \cdots \bullet \Sigma\, f : \tau'_n \bullet \mathit{body}$$

This is understandable: Function symbols generally vary more often than proof obligations. (This is discussed in detail in Section 3.1.3 and rendered in concrete Agda code in Section 5.1.) However, when users do not yet have the necessary parameters $\mathtt{p}_i$, they need to use a curried (or *bundled*) form of the module and so library developers also provide a module $\mathcal{X}$ which packs up the parameters as necessary fields within the module; i.e., $\mathcal{X}$ has the shape $\Pi^0\Sigma$ by "pushing down" the parameters into the record body. Unfortunately, there is a whole spectrum of modules $\mathcal{X}_w$ that is missing: These are the module $\mathcal{X}$ where only $w$-many of the original parameters are exposed with the remaining being packed-away into the module body; i.e., having the shape $\Pi^w\Sigma$ for $0 \leq w \leq n$ —in subsequent chapters, we refer to $w$ as "the waist" of a package former. It is tedious and error-prone to form all the $\mathcal{X}_w$ by hand; such 'unbundling' should be mechanically achievable from the completely bundled form $\mathcal{X}$. A similar issue happens when one wants to *describe a computation* using module $\mathcal{X}$, then its function symbols need to have associated syntactic counterparts —i.e., we want to interpret $\mathcal{X}$ as a $\mathcal{W}$-type instead of a $\Pi^n\Sigma$-type —; the tedium is then compounded if one considers the family $\mathcal{X}_w$. Finally, instead of combinations of $\Pi$,$\Sigma$,$\mathcal{W}$, a user may need to treat a module $\mathcal{X}$ as an arbitrary container type [**DBLP:journals/jfp/AltenkirchGHMM15**]; in which case, they will likely have to create it by hand.

This thesis aims to enhance the understanding of modules systems within dependently-typed languages by developing an in-language framework for unifying disparate presentations of what are essentially the same module. Moreover, the framework will be constructed with *practicality* in mind so that the end-result is not an unusable theoretical claim.

### 2.8.2 Objectives and Methodology

To reach a framework for the modelling of module systems for DTLs, this thesis sets a number of objectives which are described below.

#### Objective 1: Modelling Module Systems

The first objective is to actually develop a framework that models module systems —grouping mechanisms— within DTLs. The resulting framework should capture at least the expected features:

1. Namespacing, or definitions extensions —a combination of Π- and Σ-types
2. Opaque fields, or parameters —Π-types
3. Constructors, or uninterpreted identifiers —$\mathcal{W}$-types

Moreover, the resulting framework should be *practical* so as to be a usable experimentation-site for further research or immediate application —at least, in DTLs. In this thesis, we present two *declarative* approaches using meta-programming and `do`-notation.

#### Objective 2: Support Unexpected Notions of Module

The second objective is to make the resulting framework *extensible*. Users should be able to form new exotic[20] notions of grouping mechanisms *within* a DTL rather than 'stepping outside' of it and altering its interpreter —which may be a code implementation or an abstract

[20] "Exotic" in the sense that traditional module systems would not, or could not, support such constructions. For instance, some systems allow users to get the "shared structure" of two modules —e.g., for the purposes of finding a common abstract interface between them— and it does so considering *names* of symbols; i.e., an name-based intersection is formed. However, different contexts necessitate names meaningful in that context and so it would be ideal to get the shared structure by *considering* a user-provided association of "same thing, but different name" —e.g., recall that a signature has "sorts" whereas a graph has "vertices", they are the 'same thing, but have different names'.

rewrite-system. Ideally, users would be able to formulate arbitrary constructions from Universal Algebra and Category Theory. For example, given a theory —a notion of grouping— one would like to 'glue' two 'instances' along an 'identified common interface'. More concretely, we may want to treat some parameters as 'the same' and others as 'different' to obtain a new module that has copies of some parameters but not others. Moreover, users should be able to mechanically produce the necessary morphisms to make this construction into a pushout. Likewise, we would expect products, unions, intersections, and substructures of theories —when possible, and then to be constructed by users. In this thesis, we only want to provide a fixed set of meta-primitives from which usual and (un)conventional notions of grouping may be defined.

#### Objective 3: Provide a Semantics

The third objective is to provide a *concrete* semantics for the resulting framework —in contrast to the *abstract* generalised signatures semantics outlined earlier in this chapter. We propose to implement the framework in the dependently-typed functional programming language Agda, thereby automatically furnishing our syntactic constructs with semantics as Agda functions and types. This has the pleasant side-effect of making the framework accessible to future researchers for experimentation.

## 2.9 Contributions

The fulfilment of the objectives of this thesis leads to the following contributions.

1. The ability to model module systems *for* DTLs *within* DTLs
2. The ability to arbitrarily *extend* such systems by users at a high-level
3. Demonstrate that there is an expressive yet minimal set of module meta-primitives which allow common module constructions to be defined
4. Demonstrate that relationships between modules can also be *mechanically* generated.
    - ◇ In particular, if module $\mathcal{B}$ is obtained by applying a user-defined 'variational' to module $\mathcal{A}$, then the user could also enrich the child module $\mathcal{B}$ with morphisms that describe its relationships to the parent module $\mathcal{A}$.
    - ◇ E.g., if $\mathcal{B}$ is an extension of $\mathcal{A}$, then we may have a "forgetful mapping" that drops the new components; or if $\mathcal{B}$ is a 'minimal' rendition of the theory $\mathcal{A}$, then we have a "smart constructor" that forms the rich $\mathcal{A}$ by only asking the few $\mathcal{B}$ components of the user.

5. Demonstrate that there is a *practical* implementation of such a framework

6. Solve the unbundling problem: The ability to 'unbundle' module fields as if they were parameters 'on the fly'

   ◇ I.e., to transform a type of the shape $\Pi^w \Sigma$ into $\Pi^{w+k} \Sigma$, for $k \geq 0$, such that the resulting type is *as practical and as usable* as the original

7. Bring algebraic data types —i.e., *termtypes* or $\mathcal{W}$*-types*— under the umbrella of grouping mechanisms: An ADT is just a context whose symbols target the ADT 'carrier' and are not otherwise interpreted

   ◇ In particular, both an ADT and a record can be obtained from a *single* context declaration.

8. Show that common data-structures are *mechanically* the (free) termtypes of common modules.

   ◇ In particular, lists arise from modules modelling collections whereas nullables —the `Maybe` monad— arises from modules modelling pointed structures.

   ◇ Moreover, such termtypes also have a *practical* interface.

9. Finally, the resulting framework is *mostly type-theory agnostic*: The target setting is DTLs but we only assume the barebones as discussed in **??**; if users drop parts of that theory, then *only* some parts of the framework will no longer apply.

   ◇ For instance, in DTLs without a fixed-point functor the framework still 'applies', but can no longer be used to provide arbitrary algebraic data types from contexts. Instead, one could settle for the safer $\mathcal{W}$-types, if possible.

**Prerequisite of the reader**

Going forward, it is assumed that the reader is comfortable programming with Haskell, and the associated menagerie of Category Theory concepts that are usually present in the guise of Functional Programming. In particular, this includes 'practical' notions such as typeclasses and instance search, as well as 'theoretical' notions such categorial limits and colimits, lattices —a kind of category with products— and monoids —possibly in arbitrary monoidal categories, as is the case with monads.

Moreover, we assume the reader to have **actually** worked with a dependently-typed language; otherwise, it *may* be difficult to appreciate the solutions to the problems addressed in this thesis —since they could not be expressed in languages without dependent-types and are thus 'not problems'.

# 3 Motivating the problem —Examples from the Wild

*Tedium is for machines; interesting problems are for people.*

In this section, we showcase a number of problems that occur in developing libraries of code *within* dependently-typed languages. We will refer back to these real-world examples later on when developing our frameworks for reducing their tedium and size. The examples are extracted from Agda libraries focused on mathematical domains, such as algebra and category theory. It is not important to understand the application domains, but how modules are organised and used. The examples will focus on readability (sections 3.1, 3.2) and on mixing-in features to an existing module (sections 3.1.3, 3.3, 3.4). In order to make the core concepts acceptable, we will occasionally render examples using the simple algebraic structures: Magma , Semigroup, and Monoid [1].

Incidentally, the common solutions to the problems presented may be construed as **design patterns for dependently-typed programming**. Design patterns are algorithms yearning to be formalised. The power of the host language dictates whether design patterns remain as informal directions to be implemented in an ad-hoc basis then checked by other humans, or as a library methods that are written once and may be freely applied by users. For instance, the Agda `Algebra.Morphism` "library"[2] presents *only* an example(!) of the homomorphism design pattern —which shows how to form operation-preserving functions for algebraic structures. The documentation reads: `An example showing how a morphism`

[1]A *magma* `(C, ⨾)` is a set `C` and a binary operation `_⨾_ : C → C → C` on it; a *semigroup* is a magma whose operation is associative, `∀ x, y, z • (x ⨾ y) ⨾ z = x ⨾ (y ⨾ z)`; and a *monoid* is a semigroup that has a point `Id : C` acting as the identity of the binary operation: `∀ x • x ⨾ Id = x = Id ⨾ x`. For example, real numbers with subtraction `(ℝ, -)` are only a magma whereas numbers with addition `(ℝ, _+_, 0)` form a monoid. The *canonical models* of magma, semigroup, and monoid are trees (with branching), non-empty lists (with catenation), and possibly empty lists, respectively —these are discussed again in section 5.4.

[2]All references to the Agda Standard Library refer to version 0.7. The current version is 1.3, however, for the `Algebra.Morphism` library, the newer library only refactors the one monolithic homomorphism example into a fine grained hierarchy of homomorphisms. The library can be accessed at `https://github.com/agda/agda-stdlib`.

`type can be defined`. An example, rather than a library method, is all that can be done since the current implementation of Agda does not have the necessary meta-programming utilities to construct new types in a practical way —at least, not out of the box.

Chapter Contents

## 3.1 Simplifying Programs by Exposing Invariants at the Type Level

In this section, we want to discuss how "unbundled (possibly value-parameterised) presentations" can be used to simplify programs and statements about elements of shared types. In particular, this section is about "how a user may wish things were bundled" and a suggestion to "how a library designer should bundle data".

We begin with a ubqutious problem[3] that happens in practice: Given a list `[`$\texttt{x}_0$`, `$\texttt{x}_1$`, ..., `$\texttt{x}_{n-1}$`]`, how do we get the $k^{th}$ element of the list? Unless $0 \leq k < n$, we will have an error. The issue is clearly at the 'bounds', 0 and $n$, and so, for brevity, we focus on the problem of extracting the first element of a list —i.e., the first bound. The resulting unbundling solution has its own problems, so afterward, we consider how to phrase composition of programs in general and abstract that to phrasing distributivity laws. Finally, from the previous two discussions, we conclude with a promising suggestion that may improve library design.

[3]A variation of this problem is discussed in section 2.3.

### 3.1.1 Avoiding "Out-of-bounds" Errors

Let us "see the problem" by writing a function `head` that gets the first element of a list —a very useful and commonly used operation.

A list `[`$x_0$`, `$x_1$`, …, `$x_{n-1}$`]` is composed by repeatedly prepending new elements to the front of existing lists, starting from an empty list. That is, the informal notation `[`$x_0$`, `$x_1$`, …, `$x_{n-1}$`]` is represented formally as $x_0$ `::` `(`$x_1$ `::` `(`$\cdots$ `::` `(`$x_n$ `::` `[])))` using a prepending constructor `_::_` and an empty list constructor `[]`.

Lists as Algebraic Data Types

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A
```

With lists in-hand, we can try to define the `head` function.

Partially defined `head`

```
head : ∀ {A} → List A → A
head []        = {! !}
head (x :: xs) = x
```

To define `head l` for any list `l`, we consider the possible shape of the list `l`. The two possible shapes are an empty list `[]` and a prepending of an element `x` to another list `xs`. In the second case, the the list has `x` as the first element and so we yield that. Unfortunately, in the scenario of an empty list, there is no first element to return! However, `head` is typed `List A → A` and so it must somehow produce an `A` value from any given `List A` value. In general, this is not possible: If `A` is an empty type, having no values at all, then `[]` is the only possible list of `A`'s, and so `head []` is a value of `A`, which contradicts the fact that `A` is empty. Hence, either `head` remains a partially-defined function —leaving users the burden of ensuring that any call `head l` never happens with `l = []`— or one has to "add fictitious elements to every type" such as $undefined_A : A$ —thereby having no empty types at all, (roughly put, this is what Haskell does). Agda allows empty types and we can `postulate` ficitious values, or parameterise our function by a "default value", but instead we can *add the non-emptiness condition* `l ≠ []` to the type level and have it *checked at compile-time by the machine rather than by the user.*

We define the *predicate* `l ≠[]` as a data-type whose values *witness* the truth of the statement "`l` is not an empty list". As with `head`, it suffices to consdier the possible shapes of `l`. When `l` is a non-empty list `x :: xs`, then we shall include a constructor, call it, `indeed` whose type is `(x :: xs) ≠[]`; i.e., `indeed` is a 'proof' that the predicate holds for `_::_` constructions. Since `[]` is an empty list, we do not include any constructors of the type `[] ≠[]`, since that would not capture the non-emptiness predicate.

Non-emptiness Predicate

```
data _≠[] {A : Set} : List A → Set where
  indeed : ∀ {x xs} → (x :: xs) ≠[]
```

With the non-emptiness predicate/type, we can now form `head` as a totally defined function.

Non-emptiness proviso at the type level —Using an auxilary type

```
head : ∀ {A}  →  Σ l : List A • l ≠[]  →  A
head ([] , ())
head (x :: xs , indeed) = x
```

In this definition, we pattern match on the possible ways to form a list —namely, `[]` and `_::_`. In the first case, we perform *case analysis* on the shape of the proof of `[] ≠[]`, but there is no way to form such a proof and so we have "defined" the first clause of `head` using *a definition by zero-cases* on the `[] ≠[]` proof. The 'absurd pattern' `()` indicates the impossibility of a construction. The second clause is as before in the previous attempt to define `head`. This approach to "padding" the list type with auxiliary constraints *after the fact* is known as 'Σ-padding' and is discussed in section 3.1.3.

The need to introduce an auxiliary type was to "keep track" of the fact that the given list's length is not 0 and so it has an element to extract. Indeed, some popular languages have list types that "know their own length" but it is a *value field* of the type that is not observable at the type level. In a dependently-typed language, we can form a type of lists that "document the length" of the list *at the type level* —these are 'vectors'.

Exposing Information At the Type Level

```
data Vec (A : Set) : ℕ → Set where
  []   : Vec A 0
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Our type of vectors[4] is defined intentionally using the same constructor names as that of lists, which Agda allows. Notice that the first constructor is declared to be a member of the type `Vec A 0`, whereas the second declares `x :: xs` to be in `Vec A (suc n)` when `xs` is in `Vec A n`, and so `l : Vec A n` implies that the length of `l` is `n`. In particular, if `l : Vec A (suc n)` then `l` has a postive length and so is non-empty; i.e., non-emptiness can be expressed directly in the type of `l`.

[4]The definition of this type, and the subsequent `head` function, have been discussed in section 2.4.2, in the introduction to dependently-typed programming with Agda.

Non-emptiness proviso at the type level

```
head′ : ∀ {A n} → Vec A (suc n) → A
head′ (x :: xs) = x
```

As usual, this function is defined on the shape of its argument. Since its argument is a value of `Vec A (suc n)`, only the prepending constructor `_::_` of the `Vec` type is possible, and so the definition has only one clause; from which we immediately extract an `A`-value, namely `x`.

Before we conclude this section, it is interesting to note that we could have used a type `Vec′ : (A : Set) (empty-or-not : 𝔹) → Set` that only documents whether a list is empty or not. However, this option is less useful than the one that keeps track of a list's length. Indeed, a list's length is useful as a "quick sanity check" when defining operations on lists, and so having this simple correctness test embedded at the (*machine-checkable!*) type level results in a form of "simple specfication" of functions. For example, the types of common list operations can have some of their behaviour reflected in their type via lengths of lists:

Simple Partial Specfications of List Operations

```
{- Neither length nor value type changes -}
reverse : ∀ {A n} → Vec A n → Vec A n

{- Only the type changes, the length stays the same -}
map     : ∀ {A B n} → (A → B) → Vec A n → Vec B n

{- Length of the result is sum of lengths of inputs -}
_++_    : ∀ {A m n} → Vec A m → Vec A n → Vec A (m + n)
```

In theory, lists and vectors are the same[5] —where the latter are essentially lists indexed by their lengths. In practice, however, the additional length information stated up-front as an integral part of the data structure makes it not only easier to write programs that would otherwise be awkward or impossible[6] in the latter case. For instance, above we demonstrated that the function `head`, which extracts the first element of a non-empty list, not only has a difficult type to read, but also requires an auxiliary relation/type in order to be expressed. In contrast, the vector variant has a much simpler type with the non-emptiness proviso expressed by requesting a positive length.

[5]Formally, one could show, for instance, that every list corresponds to a vector, `List X` ≅ `(Σ n : ℕ • Vec X n)`. Informally, any list $x_1$ :: $x_2$ :: … :: $x_n$ :: `[]` can be treated as a vector (since we are using the same *overloaded* constructors for both types) of *length* `n`; conversely, given a vector in `Vec X n`, we "forget" the length to obtain a list.

[6]For example, to find how many elements are in a list, a function `length : ∀ {A} → List A → ℕ` must "walk along each prepending constructor until it reaches the empty constructor" and so it requires as many steps to compute as there are elements in the list. As such, it is impossible to write a function that requires a constant amount of steps to obtain the length of a list. In contrast, a function `length : ∀ {A n} → Vec A n → ℕ` requires *zero steps* to compute its result —namely, `length {A} {n} l = n`— and so this function, for vectors, is rather facetious.

It seems that vectors are the way to go —but that depends on where one is *going*. For example, if we want to keep only elements of a vector that satisfy a predicate `p`, as shown below. To type such an operation we need to either know how many elements `m` satisfy the predicate ahead of time, and so the return type is `Vec A m`; or we 'Σ-pad' the length parameter to essentially demote it from the type level to the body level of the program.

Eek!

```
filter : ∀ {A n} → (A → 𝔹) → Vec A n → Σ m : ℕ • Vec A m
filter p [] = 0 , []
filter p (x :: xs) with p x
...| true  = let (m , ys) = filter p xs in 1 + m , x :: ys
...| false = filter p xs
```

*Equivalent structures, but different usability profiles.*

### 3.1.2 "Obviously sharing the same type" requires 'do-nothing' conversion functions! —Unbundling

The phenomenon of exposing attributes at the type level to gain flexibility applies not only to derived concepts such as non-emptiness, but also to explicit features of a datatype. A common scenario is when two instances of an algebraic structure share the same carrier and thus it is reasonable to connect the two somehow by a coherence axiom. That is, the "same problem" arises when, for example, discussing the interaction between sequential program composition `_⨾_` and parallel program composition `_||_`: The *simultaneous* execution of programs `P`-then-`P′` and `Q`-then-`Q′` results in the same behaviour as the *sequential* execution of `P`-and-simultaneously-`Q` then `P′`-and-simultaneously-`Q′`. That is, `(P ⨾ P′) || (Q ⨾ Q′) = (P || Q) ⨾ (P′ ⨾ Q′)`. But for this equation to be well-typed, we need to *know* that the composition operators work on the *same kind* of programs phrases —it is surprisingly not enough to know that each combines certain kinds of program phrases that happen to be the same kind.

For brevity, rather than consider program language phrases and operators on them, let us abstract and consider what is perhaps the most popular instance of structure-sharing known to many from childhood, in the setting of rings: We have an additive structure `(R, +)` and a multiplicative structure `(R, ×)` on the same underlying set `R`, and their interaction is dictated by distributivity axioms, such as $a \times (b + c) = (a \times b) + (a \times c)$. As with `head` above, depending on which features of the structure are exposed upfront, such axioms may be either difficult to express or relatively easy. Below are the two possible ways to present a structure admiting a type and a binary operation on that type.

To bundle or to not bundle?

```
{- A magma₀ is a pair ⟨C, op⟩ of a type C and an operation ‘op’ on that type -}
record Magma₀ : Set₁ where
  constructor ⟨_,_⟩₀
  field
    Carrier : Set
    _⨾_ : Carrier → Carrier → Carrier

{- A magma₁ “on” a given type C is a one-tuple ⟨op⟩
   consisting of a binary operation on that type -}
record Magma₁ (Carrier : Set) : Set₁ where
  constructor ⟨_⟩₁
  field
    _⨾_ : Carrier → Carrier → Carrier
```

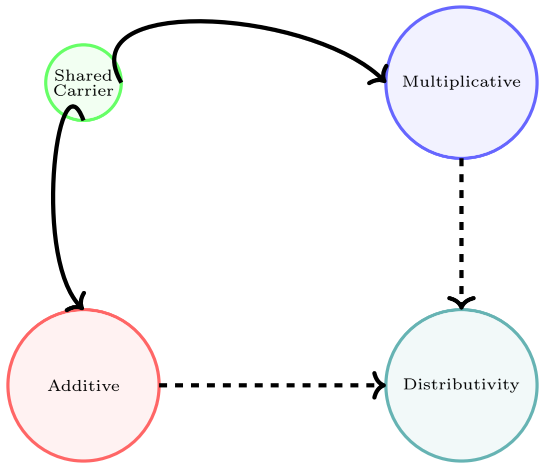
In **theory**, parameterised structures are no different from their unparameterised, or “bundled”, counterparts. Indeed, we can easily prove `Magma₀ ≅ (Σ C : Set • Magma₁ C)` by “packing away the parameters” and `∀ (C : Set) → Magma₁ C ≅ (Σ M : Magma₀ • M.Carrier ≡ C)` by “abstracting a field as if it were a parameter” —this is known as ‘Σ-padding’. Below is a proof in Agda of the first isomorphism; the other isomorphism is proven just as easily but suffers from excess noise introduced by the Σ-padding, namely extra phrases “ `, refl` ” that serve to keep track of important facts, but are otherwise unhelpful. The proofs generalise easily on a case-by-case basis to other kinds of structures, but they cannot be proven internally to Agda in full generality.

To bundle or to not bundle?

```
{- Abstract out a field -}
to    :  Magma₀  →  Σ C : Set • Magma₁ C
to M = Magma₀.Carrier M , ⟨ Magma₀._⨾_ M ⟩₁

{- Pack away a parameter -}
from   :  Σ C : Set • Magma₁ C  →  Magma₀
from (C , ⟨ _⨾_ ⟩₁) = ⟨ C , _⨾_ ⟩₀

-- These are inverse by “definition chasing” (normalisation).

to∘from : ∀ M → from (to M) ≡ M
to∘from ⟨ Carrier , _⨾_ ⟩₀ = refl

from∘to : ∀ M → to (from M) ≡ M
from∘to (C , ⟨ _⨾_ ⟩₁) = refl
```

Let us consider *using* the first presentation. When structures “pack away” all their features, the simple distributivity property becomes a bit of a challenge to write and to read.

Figure 3.1: Bundled forms require (curved) coercions



Distributivity is Difficult to Express

```
record Distributivity₀ (Additive Multiplicative : Magma₀) : Set₁ where

  open Magma₀ Additive       renaming (Carrier to R₊; _⨾_ to _+_)
  open Magma₀ Multiplicative renaming (Carrier to R×; _⨾_ to _×_)

  field shared-carrier :  R₊ ≡ R×

  coe× :R₊ →R×
  coe× = subst id shared-carrier

  coe₊ :R× →R₊
  coe₊ = subst id (sym shared-carrier)

  field distribute₀ : ∀ {a : R×} {b c : R₊}
                    →    a × coe× (b + c)
                      ≡ coe× (coe₊(a × coe× b) + coe₊(a × coe× c))
```

It is a bit of a challenge to understand the type of distribute₀. Even though the carriers of the structures are propositionally equal, $R_+ \equiv R_\times$, they are not the same by definition —the notion of equality was defined in section 2.4.3. As such, we are forced to "coe"rce back and forth; leaving the distributivity axiom as an exotic property of addition, multiplication, and coercions. Even worse, without the cleverness of declaring two coercion helpers, the typing of distribute₀ would have been so large and confusing that the concept would be rendered near useless. In particular, the **cleverness** is captured by the solid curved arrows in the following *informal* diagram —where the dashed lines denote inclusions or dependency relationships.

Again, in theory, parameterised structures are no different from their unparameterised, or “bundled”, counterparts. However, in **practice**, even when multiple presentations of an idea are *equivalent* in some sense, there may be specfic presentations that are *useful* for particular purposes[7]. That is, in a depeendely-typed language, equivalence of structures and their usability profiles do not necessairly go hand-in-hand. Indeed, below we can phrase the distributivity axiom nearly as it was stated informally earlier since the shared carrier is declared upfront.

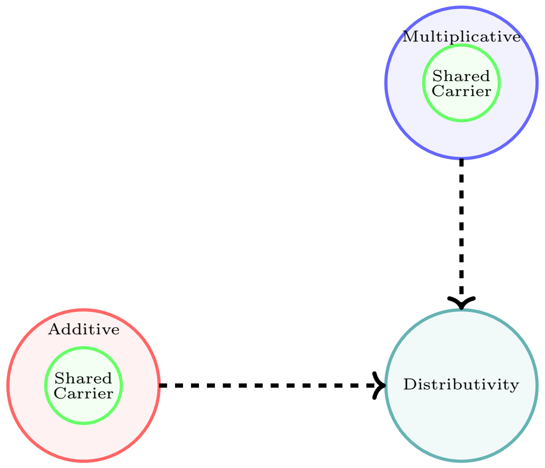
Distributivity is Expressed Easily with Unbundled Structures

```
{- A magma “on” a given type is a binary operation on that type -}
record Magma₁ (Carrier : Set) : Set₁ where
  field
    _⨾_      : Carrier → Carrier → Carrier

record Distributivity₁
    (R : Set) {- The shared carrier -}
    (Additive Multiplicative : Magma₁ R)  : Set₁ where

  open Magma₁ Additive       renaming (_⨾_ to _+_)
  open Magma₁ Multiplicative renaming (_⨾_ to _×_)

  field distribute₁ : ∀ {a b c : R} →  a × (b + c) ≡ (a × b) + (a × c)
```

In contrast to the bundled definition of magmas, this form requires no cleverness to form coercion helpers, and is closer to the informal and usual distributivity statement. The **lack** of the aforementioned cleverness is captured by the following diagram: There are no solid curved arrows that *indicate how the shared component is to be found*; instead, the shared component is explicit.

[7]In theory, numbers can be presented equivalently using Arabic or Roman numerals. In practice, doing arithmetic is much more efficient using the former presentation.

Figure 3.2: Unbundled forms have shared components stated explicitly (as parameters)



By the same arguments above, the simple statement relating the two units of a ring $1 \times r + 0 = r$ —or any units of monoids sharing the same carrier— is easily phrased using an unbundled presentation and would require coercions otherwise. We invite the reader to pause at this moment to appreciate the difficulty in simply expressing this property.

**Unbundling Design Pattern**

If a feature of a class is shared among instances, then use an unbundled form of the class to avoid "coercion hell". See Sections 3.1.3, 2.8.1, 5.1.

### 3.1.3 From `Is`$\mathcal{X}$ to $\mathcal{X}$ —Packing away components

The distributivity axiom from required an unbundled structure *after* a completely bundled structure was initially presented. Usually structures are rather large and have libraries built around them, so building and using an alternate form is not practical. However, multiple forms are usually desirable.

To accommodate the need for both forms of structure, Agda's Standard Library begins with a type-level predicate such as `IsSemigroup` below, then packs that up into a record. Here is an instance, along with comments from the library.

From Is$\mathcal{X}$ to $\mathcal{X}$ —where $\mathcal{X}$ is Semigroup

```
-- Some algebraic structures (not packed up with sets, operations, etc.)
record IsSemigroup {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                   (· : Op₂ A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    isEquivalence : IsEquivalence ≈
    assoc         : Associative ·
    ·-cong        : · Preserves₂ ≈ ⟶ ≈ ⟶ ≈
```

From Is$\mathcal{X}$ to $\mathcal{X}$ —where $\mathcal{X}$ is Semigroup

```
-- Definitions of algebraic structures like monoids and rings
-- (packed in records together with sets, operations, etc.)
record Semigroup c ℓ : Set (suc (c ⊔ ℓ)) where
  infixl 7 _·_
  infix  4 _≈_
  field
    Carrier     : Set c
    _≈_         : Rel Carrier ℓ
    _·_         : Op₂ Carrier
    isSemigroup : IsSemigroup _≈_ _·_
```

If we refer to the former as `Is`$\mathcal{X}$ and the latter as $\mathcal{X}$, then we can see similar instances in the standard library for $\mathcal{X}$ being: `Monoid`, `Group`, `AbelianGroup`, `CommutativeMonoid`, `SemigroupWithoutOne`, `NearSemiring`, `Semiring`, `CommutativeSemiringWithoutOne`, `CommutativeSemiring`, `CommutativeRing`.

It thus seems that to present an idea $\mathcal{X}$, we require the same amount of space to present it unpacked or packed, and so doing both **duplicates the process** and only hints at the underlying principle: From `Is`$\mathcal{X}$ we pack away the carriers and function symbols to obtain $\mathcal{X}$. The converse approach, starting from $\mathcal{X}$ and going to `Is`$\mathcal{X}$ is not practical, as it leads to numerous unhelpful reflexivity proofs —c.f., the `indeed` proof of the `_≠[]` type for lists, from section 3.1.1.

**Predicate Design Pattern**

Present a concept $\mathcal{X}$ first as a predicate `Is`$\mathcal{X}$ on types and function symbols, then as a type $\mathcal{X}$ consisting of types, function symbols, and a proof that together they satisfy the `Is`$\mathcal{X}$ predicate.

**Σ-Padding Anti-Pattern**: Starting from a bundled up type $\mathcal{X}$ consisting of types, function symbols, and how they interact, one may form the type `Σ X :` $\mathcal{X}$ `•` $\mathcal{X}$`.f X ≡` $\mathbf{f}_0$ to specialise the feature $\mathcal{X}$`.f` to the particular choice $\mathbf{f}_0$. However, nearly all uses of this type will be of the form `(X , refl)` where the proof is unhelpful noise.

Since the standard library uses the predicate pattern, $\texttt{Is}\mathcal{X}$, which requires all sets and function symbols, the Σ-padding anti-pattern becomes a necessary evil. Instead, it would be preferable to have the family $\mathcal{X}_i$ which is the same as $\texttt{Is}\mathcal{X}$ but only takes $i$-many elements —c.f., $\texttt{Magma}_0$ and $\texttt{Magma}_1$ above. However, writing these variations and the necessary functions to move between them is not only tedious but also error prone. Later on, also demonstrated in [**DBLP:conf/gpce/Al-hassyCK19**], we shall show how the bundled form $\mathcal{X}$ acts as ***the*** definition, with other forms being derived-as-needed.

In summary, as the previous two discussions have shown, bundled presentations (as in $\mathcal{X}_0$) suffer from the inability to declare *shared* components between structures —thereby necessitating some form of Σ-padding— and makes working with shared components is non-trivial due to the need to rewrite along propositional equalities, as was the case with simply stating the distributivity law using $\texttt{Magma}_0$. Another problem with fully bundled structures is that accessing deeply nested components requires length projection paths, which is not only cumbersome but also exposes the hierarchical design of the structure, thereby limiting library designers from reorganising such hierarchies in the future. In constrast, unbundled presentations (as in $\mathcal{X}_n$, for $n$ the number of sort and function symbols of the structure) are flexible in theory, but in practice one must enumerate all components to actually state and apply results about such structures.

Incidentally, the particular choice $\mathcal{X}_1$, a predicate on one carrier, deserves special attention. In Haskell, instances of such a type are generally known as *typeclass instances* and $\mathcal{X}_1$ is known as a *typeclass*. As discussed earlier, in Agda, we may mark such implementations for instance search using the keyword `instance`.

**Typeclass Design Pattern**

Present a concept $\mathcal{X}$ as a unary predicate $\mathcal{X}_1$ that associates functions and properties with a given type. Then, mark all implementations with `instance` so that arbitrary $\mathcal{X}$-terms may be written without having to specify the particular instance.

As discussed in section 2.5, when there are multiple instance of an $\mathcal{X}$-structure on a particular type, only one of them may be marked for instance search in a given scope.

*Type Classes for Mathematics in Type Theory* [**typeclasses_for_maths**] discusses the numerous problems of bundled presentations as well as the issues of unbundled presentations and settles on using typeclasses along with their tremnously useful instance search mechanism. Since we view $\mathcal{X}_1$ as a particular choice in the family $(\mathcal{X}_w)_{w \in \mathbb{N}}$, our approach is to instead have library designers define $\mathcal{X}_0$ and let users *easily, mechanically, declaratvely,* produce $\mathcal{X}_w$ for any 'parameterisation waist' $w : \mathbb{N}$. This idea is implemented for Agda, as an in-language library, and discussed in chapter 5.

Notice that to phrase the distirbutivy law we assigned superficial renamings, aliases, to the prototypical binary operation `_⨾_` so that we may phrase the distributivity axiom in its expected notational form. This leads us to our next topic of discussion.

## 3.2 Renaming

The use of an idea is generally accompanied with particular notation that is accepted by its primary community. Even though the choice of bound names it theoretically irrelevant, certain communities would consider it unacceptable to deviate from convention. Here are a few examples:

$x(f)$ Using $x$ as a *function* and $f$ as an *argument.*; likewise $\frac{\partial x}{\partial f}$.
With the exception of discussions involving the Yoneda Lemma, or continuations, such a notation is simply *'wrong'*.

$a \times a = a$ An idempotent operation denoted by multiplication; likewise for commutative operations. It is more common to use addition or join, '$\sqcup$'.

$0 \times a \approx a$ The identity of "multiplicative symbols" should never resemble '0'; instead it should resemble '1' or, at least, '$e$' —the standard abbreviation of the influential algebraic works of German authors who used "Einheit" which means "identity".

$f + g$ Even if monoids are defined with the prototypical binary operation denoted '+', it would be *'wrong'* to continue using it to denote functional composition. One would need to introduce the new name '$\circ$' or, at least, '$\cdot$'.

From the few examples above, it is immediate that to even present a prototypical notation for an idea, one immediately needs auxiliary notation when specialising to a particular instance. For example, to use 'additive symbols' such as $+, \sqcup, \oplus$ to denote an arbitrary binary operation leads to trouble in the function composition instance above, whereas using 'multiplicative symbols' such as $\times, , *$ leads to trouble in the idempotent case above. Regardless of prototypical choices, there will always be a need to rename.

**Renaming Design Pattern**

Use superficial aliases to better communicate an idea; especially so, when the topic domain is specialised.

Let's now turn to examples of renaming from three libraries:

1. Agda's "standard library" [**agda_std_lib**],
2. The "RATH-Agda" library [**RATH**], and
3. A recent "agda-categories" library [**copumpkin**].

Each will provide a workaround to the problem of renaming. In particular, the solutions are, respectively:

1. **Rename as needed.**
   - ◇ There is no systematic approach to account for the many common renamings.
   - ◇ Users are encouraged to do the same, since the standard library does it this way.
2. **Pack-up the *common* renamings as modules, and invoke them when needed.**
   - ◇ Which renamings are provided is left at the discretion of the designer —even 'expected' renamings may not be there since, say, there are too many choices or insufficient man power to produce them.
   - ◇ The pattern to pack-up renamings leads nicely to consistent naming.
3. **Names don't matter.**
   - ◇ Users of the library need to be intimately connected with the Agda definitions and domain to use the library.
   - ◇ Consequently, there are many inconsistencies in naming.

The `open` ⋯ `public` ⋯ `renaming` ⋯ pattern shown below will be presented later, section 4.3, as a library method.

### 3.2.1 Renaming Problems from Agda's Standard Library

Here are four excerpts from Agda's standard library, notice how the prototypical notation for monoids is renamed **repeatedly** *as needed.* Sometimes it is relabelled with additive symbols, other times with multiplicative symbols. The content itself is not important, instead the focus is on the renaming that takes place —as such, the fontsize is intentionally tiny.

**Additive Renaming —IsNearSemiring**

```
record IsNearSemiring {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                      (+ * : Op₂ A) (0# : A) : Set (a ⊔ ℓ)
                     ↪  where
  open FunctionProperties ≈
  field
    +-isMonoid    : IsMonoid ≈ + 0#
    *-isSemigroup : IsSemigroup ≈ *
    distribʳ      : * DistributesOverʳ +
    zeroˡ         : LeftZero 0# *

  open IsMonoid +-isMonoid public
         renaming ( assoc       to +-assoc
                  ; ·-cong      to +-cong
                  ; isSemigroup to +-isSemigroup
                  ; identity    to +-identity
                  )

  open IsSemigroup *-isSemigroup public
         using ()
         renaming ( assoc    to *-assoc
                  ; ·-cong   to *-cong
                  )
```

**Additive Renaming Again —IsSemiringWithoutOne**

```
record IsSemiringWithoutOne {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                            (+ * : Op₂ A) (0# : A) : Set (a ⊔
                           ↪  ℓ)
 where
  open FunctionProperties ≈
  field
    +-isCommutativeMonoid : IsCommutativeMonoid ≈ + 0#
    *-isSemigroup         : IsSemigroup ≈ *
    distrib               : * DistributesOver +
    zero                  : Zero 0# *

  open IsCommutativeMonoid +-isCommutativeMonoid public
         hiding (identityˡ)
         renaming ( assoc       to +-assoc
                  ; ·-cong      to +-cong
                  ; isSemigroup to +-isSemigroup
                  ; identity    to +-identity
                  ; isMonoid    to +-isMonoid
                  ; comm        to +-comm
                  )

  open IsSemigroup *-isSemigroup public
         using ()
         renaming ( assoc       to *-assoc
                  ; ·-cong      to *-cong
                  )
```

**Additive Renaming a 3rd Time and Multiplicative Renaming —IsSemiringWithoutAnnihilatingZero**

```
record IsSemiringWithoutAnnihilatingZero
       {a ℓ} {A : Set a} (≈ : Rel A ℓ)
       (+ * : Op₂ A) (0# 1# : A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    +-isCommutativeMonoid : IsCommutativeMonoid ≈ + 0#
    *-isMonoid            : IsMonoid ≈ * 1#
    distrib               : * DistributesOver +

  open IsCommutativeMonoid +-isCommutativeMonoid public
       hiding (identityˡ)
       renaming ( assoc       to +-assoc
                ; ·-cong      to +-cong
                ; isSemigroup to +-isSemigroup
                ; identity    to +-identity
                ; isMonoid    to +-isMonoid
                ; comm        to +-comm
                )

  open IsMonoid *-isMonoid public
       using ()
       renaming ( assoc       to *-assoc
                ; ·-cong      to *-cong
                ; isSemigroup to *-isSemigroup
                ; identity    to *-identity
                )
```

**Additive Renaming a 4th Time and Second Multiplicative Renaming —IsRing**

```
record IsRing
       {a ℓ} {A : Set a} (≈ : Rel A ℓ)
       (_+_ _*_ : Op₂ A) (-_ : Op₁ A) (0# 1# : A) : Set (a
       ↪  ⊔ ℓ)
 where
  open FunctionProperties ≈
  field
    +-isAbelianGroup : IsAbelianGroup ≈ _+_ 0# -_
    *-isMonoid       : IsMonoid ≈ _*_ 1#
    distrib          : _*_ DistributesOver _+_

  open IsAbelianGroup +-isAbelianGroup public
       renaming ( assoc                to +-assoc
                ; ·-cong               to +-cong
                ; isSemigroup          to +-isSemigroup
                ; identity             to +-identity
                ; isMonoid             to +-isMonoid
                ; inverse              to -CONVERSEinverse
                ; ⁻¹-cong               to -CONVERSEcong
                ; isGroup              to +-isGroup
                ; comm                 to +-comm
                ; isCommutativeMonoid to
               ↪  +-isCommutativeMonoid
                )

  open IsMonoid *-isMonoid public
       using ()
       renaming ( assoc       to *-assoc
                ; ·-cong      to *-cong
                ; isSemigroup to *-isSemigroup
                ; identity    to *-identity
                )
```

At first glance, one solution would be to package up these renamings into helper modules. For example, consider the setting of monoids.

**Original —Prototypical— Notations**

```
record IsMonoid {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                (· : Op₂ A) (ε : A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    isSemigroup : IsSemigroup ≈ ·
    identity    : Identity ε ·

record IsCommutativeMonoid {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                           (_·_ : Op₂ A) (ε : A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    isSemigroup : IsSemigroup ≈ _·_
    identityˡ   : LeftIdentity ε _·_
    comm        : Commutative _·_

   ⋮
  isMonoid : IsMonoid ≈ _·_ ε
  isMonoid = record { ⋯ }
```

Renaming Helper Modules

```
module AdditiveIsMonoid {a ℓ} {A : Set a} {≈ : Rel A ℓ}
               {_·_ : Op₂ A} {ε : A} (+-isMonoid : IsMonoid ≈ _·_ ε)  where

   open IsMonoid +-isMonoid public
         renaming ( assoc       to +-assoc
                  ; ·-cong      to +-cong
                  ; isSemigroup to +-isSemigroup
                  ; identity    to +-identity
                  )

module AdditiveIsCommutativeMonoid {a ℓ} {A : Set a} {≈ : Rel A ℓ}
               {_·_ : Op₂ A} {ε : A} (+-isCommutativeMonoid : IsMonoid ≈ _·_ ε)
               ↪  where

   open AdditiveIsMonoid (CommutativeMonoid.isMonoid +-isCommutativeMonoid) public
   open IsCommutativeMonoid +-isCommutativeMonoid public using ()
      renaming ( comm to +-comm
               ; isMonoid to +-isMonoid)
```

However, one then needs to make similar modules for *additive notation* for `IsAbelianGroup`, `IsRing`, `IsCommutativeRing`, …. Moreover, this still invites repetition: Additional notations, as used in `IsSemiring`, would require additional helper modules.

More Necessary Renaming Helper Modules

```
module MultiplicativeIsMonoid {a ℓ} {A : Set a} {≈ : Rel A ℓ}
               {_·_ : Op₂ A} {ε : A} (*-isMonoid : IsMonoid ≈ _·_ ε)  where

   open IsMonoid *-isMonoid public
         renaming ( assoc       to *-assoc
                  ; ·-cong      to *-cong
                  ; isSemigroup to *-isSemigroup
                  ; identity    to *-identity
                  )
```

Unless carefully organised, such notational modules would bloat the standard library, resulting in difficulty when navigating the library. As it stands however, the new algebraic structures appear large and complex due to the “renaming hell” encountered to provide the expected conventional notation.

### 3.2.2 Renaming Problems from the RATH-Agda Library

The impressive Relational Algebraic Theories in Agda library takes a disciplined approach: Copy-paste notational modules, possibly using a find-replace mechanism to vary the notation. The use of a find-replace mechanism leads to consistent naming across different notations.

Quoting the library, *For contexts where calculation in different setoids is necessary, we provide "decorated" versions of the* ***Setoid′*** *and* ***SetoidCalc*** *interfaces:*

Seotoid𝒟 Renamings —𝒟decorated Synonyms

```
module SetoidA {i j : Level} (S : Setoid i j) = Setoid′ S renaming
    ( ℓ to ℓA ; Carrier to A₀ ; _≈_ to _≈A_ ; ≈-isEquivalence to ≈A-isEquivalence
    ; ≈-isPreorder to ≈A-isPreorder ; ≈-preorder to ≈A-preorder
    ; ≈-indexedSetoid to ≈A-indexedSetoid
    ; ≈-refl to ≈A-refl ; ≈-reflexive to ≈A-reflexive ; ≈-sym to ≈A-sym
    ; ≈-trans to ≈A-trans ; ≈-trans₁ to ≈A-trans₁ ; ≈-trans₂ to ≈A-trans₂
    ; _⟨≈≈⟩_ to _⟨≈A≈⟩_ ; _⟨≈≈˘⟩_ to _⟨≈A≈˘⟩_ ; _⟨≈˘≈⟩_ to _⟨≈A˘≈⟩_
    ; _⟨≈˘≈˘⟩_ to _⟨≈A˘≈˘⟩_; _⟨≡≈⟩_ to _⟨≡≈A⟩_ ; _⟨≡≈˘⟩_ to _⟨≡≈A˘⟩_
    ; _⟨≡˘≈⟩_ to _⟨≡˘≈A⟩_ ; _⟨≡˘≈˘⟩_ to _⟨≡˘≈A˘⟩_ ; _⟨≈≡⟩_ to _⟨≈A≡⟩_
    ; _⟨≈≡˘⟩_ to _⟨≈A≡˘⟩_ ; _⟨≈˘≡⟩_ to _⟨≈A˘≡⟩_ ; _⟨≈˘≡˘⟩_ to _⟨≈A˘≡˘⟩_
    )

module SetoidB {i j : Level} (S : Setoid i j) = Setoid′ S renaming
    ( ℓ to ℓB ; Carrier to B₀ ; _≈_ to _≈B_ ; ≈-isEquivalence to ≈B-isEquivalence
    ; ≈-isPreorder to ≈B-isPreorder ; ≈-preorder to ≈B-preorder
    ; ≈-indexedSetoid to ≈B-indexedSetoid
    ; ≈-refl to ≈B-refl ; ≈-reflexive to ≈B-reflexive ; ≈-sym to ≈B-sym
    ; ≈-trans to ≈B-trans ; ≈-trans₁ to ≈B-trans₁ ; ≈-trans₂ to ≈B-trans₂
    ; _⟨≈≈⟩_ to _⟨≈B≈⟩_ ; _⟨≈≈˘⟩_ to _⟨≈B≈˘⟩_ ; _⟨≈˘≈⟩_ to _⟨≈B˘≈⟩_
    ; _⟨≈˘≈˘⟩_ to _⟨≈B˘≈˘⟩_ ; _⟨≡≈⟩_ to _⟨≡≈B⟩_ ; _⟨≡≈˘⟩_ to _⟨≡≈B˘⟩_
    ; _⟨≡˘≈⟩_ to _⟨≡˘≈B⟩_ ; _⟨≡˘≈˘⟩_ to _⟨≡˘≈B˘⟩_ ; _⟨≈≡⟩_ to _⟨≈B≡⟩_
    ; _⟨≈≡˘⟩_ to _⟨≈B≡˘⟩_ ; _⟨≈˘≡⟩_ to _⟨≈B˘≡⟩_ ; _⟨≈˘≡˘⟩_ to _⟨≈B˘≡˘⟩_
    )

module SetoidC {i j : Level} (S : Setoid i j) = Setoid′ S renaming
    ( ℓ to ℓC ; Carrier to C₀ ; _≈_ to _≈C_ ; ≈-isEquivalence to ≈C-isEquivalence
    ; ≈-isPreorder to ≈C-isPreorder ; ≈-preorder to ≈C-preorder
    ; ≈-indexedSetoid to ≈C-indexedSetoid
    ; ≈-refl to ≈C-refl ; ≈-reflexive to ≈C-reflexive ; ≈-sym to ≈C-sym
    ; ≈-trans to ≈C-trans ; ≈-trans₁ to ≈C-trans₁ ; ≈-trans₂ to ≈C-trans₂
    ; _⟨≈≈⟩_ to _⟨≈C≈⟩_ ; _⟨≈≈˘⟩_ to _⟨≈C≈˘⟩_ ; _⟨≈˘≈⟩_ to _⟨≈C˘≈⟩_
    ; _⟨≈˘≈˘⟩_ to _⟨≈C˘≈˘⟩_ ; _⟨≡≈⟩_ to _⟨≡≈C⟩_ ; _⟨≡≈˘⟩_ to _⟨≡≈C˘⟩_
    ; _⟨≡˘≈⟩_ to _⟨≡˘≈C⟩_ ; _⟨≡˘≈˘⟩_ to _⟨≡˘≈C˘⟩_ ; _⟨≈≡⟩_ to _⟨≈C≡⟩_
    ; _⟨≈≡˘⟩_ to _⟨≈C≡˘⟩_ ; _⟨≈˘≡⟩_ to _⟨≈C˘≡⟩_ ; _⟨≈˘≡˘⟩_ to _⟨≈C˘≡˘⟩_
    )
```

This keeps going to cover the alphabet `SetoidD`, `SetoidE`, `SetoidF`, …, `SetoidZ` then we shift to subscripted versions `Setoid`$_0$, `Setoid`$_1$, …, `Setoid`$_4$.

Next, RATH-Agda shifts to the need to calculate with setoids:

SeotoidCalc𝒟 Renamings —𝒟decorated Synonyms

```
module SetoidCalcA {i j : Level} (S : Setoid i j) where
  open SetoidA S public
  open SetoidCalc S public renaming
    ( _QED to _QEDA
    ; _≈⟨_⟩_ to _≈A⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈A˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈A≡⟨_⟩_
    ; _≈⟨⟩_ to _≈A⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈A≡˘⟨_⟩_
    ; ≈-begin_ to ≈A-begin_
    )
module SetoidCalcB {i j : Level} (S : Setoid i j) where
  open SetoidB S public
  open SetoidCalc S public renaming
    ( _QED to _QEDB
    ; _≈⟨_⟩_ to _≈B⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈B˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈B≡⟨_⟩_
    ; _≈⟨⟩_ to _≈B⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈B≡˘⟨_⟩_
    ; ≈-begin_ to ≈B-begin_
    )
module SetoidCalcC {i j : Level} (S : Setoid i j) where
  open SetoidC S public
  open SetoidCalc S public renaming
    ( _QED to _QEDC
    ; _≈⟨_⟩_ to _≈C⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈C˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈C≡⟨_⟩_
    ; _≈⟨⟩_ to _≈C⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈C≡˘⟨_⟩_
    ; ≈-begin_ to ≈C-begin_
    )
```

This keeps going to cover the alphabet `SetoidCalcD, SetoidCalcE, SetoidCalcF, ...,` `SetoidCalcZ` then we shift to subscripted versions $\texttt{SetoidCalc}_0$, $\texttt{SetoidCalc}_1$, ..., $\texttt{SetoidCalc}_4$. If we ever have more than 4 setoids in hand, or prefer other decorations, then we would need to produce similar helper modules.

Each `Setoid`𝒳𝒳𝒳 takes 10 lines, for a total of at-least 600 lines!

Indeed, such renamings bloat the library, but, unlike the Standard Library, they allow new records to be declared easily —"renaming hell" has been deferred from the user to the library designer. However, later on, in `Categoric.CompOp`, we see the variations `LocalEdgeSetoid`𝒟 and `LocalSetoidCalc`𝒟 where decoration 𝒟 ranges over $_0$, $_1$, $_2$, $_3$, $_4$, `R`. The inconsistency in not providing the other decorations used for `Setoid`𝒟 earlier is understandable: These take time to write and maintain.

### 3.2.3 Renaming Problems from the Agda-categories Library

With RATH-Agda's focus on notational modules at one end of the spectrum, and the Standard Library's casual do-as-needed in the middle, it is inevitable that there are other equally popular libraries at the other end of the spectrum. The Agda-categories library seemingly ignored the need for meaningful names altogether. Below are a few notable instances.

◇ Functors have fields named $\texttt{F}_0$, $\texttt{F}_1$, `F-resp-≈`, ....

    - This could be considered reasonable even if one has a functor named `G`.
    - This leads to expressions such as `< F.F`$_0$ `, G.F`$_0$ `>`.
    - Incidentally, and somewhat inconsistently, a `Pseudofunctor` has fields `P`$_0$`,` `P`$_1$`,` `P-homomophism` —where the latter is documented *P preserves* $\simeq$.

  On the opposite extreme, RATH-Agda's focus on naming has its functor record with fields named `obj, mor, mor-cong` instead of `F`$_0$`,` `F`$_1$`,` `F-resp-`$\approx$ —which refer to a functor's "obj"ect map, "mor"phism map, and the fact that the "mor"phism map is a "cong"ruence.

◇ Such lack of concern for naming might be acceptable for well-known concepts such as functors, where some communities use `F`$_i$ to denote the object/0-cell or morphism/1-cell operations. However, considering subcategories one sees field names `U, R, Rid, _∘R_` which are wholly unhelpful. Instead, more meaningful names such as `embed, keep, id-kept, keep-resp-∘` could have been used.

◇ The `Iso, Inverse,` and `NaturalIsomorphism` records have fields `to / from, f / f`$^{-1}$, and `F ⇒ G / F ⇐ G`, respectively.

  Even though some of these build on one another, with Agda's namespacing features, all "forward" and "backward" morphism fields could have been named, say, `to` and `from`. The naming may not have propagated from `Iso` to other records possibly due to the low priority for names.

  From a usability perspective, projections like `f` are reminiscent of the OCaml community and may be more acceptable there. Since Agda is more likely to attract Haskell programmers than OCaml ones, such a particular projection seems completely out of place. Likewise, the field name `F ⇒ G` seems only appropriate if the functors involved happen to be named `F` and `G`.

  These unexpected deviations are not too surprising since the Agda-categories library seems to give names no priority at all. Field projections are treated little more than classic array indexing with numbers.

By largely avoiding renaming, Agda-categories has no "renaming hell" anywhere at the heavy price of being difficult to read: Any attempt to read code requires one to "squint away" the numerous projections to "see" the concepts of relevance. Consider the following excerpt.

Symbol Soup

```
helper : ∀ {F : Functor (Category.op C) (Setoids ℓ e)}
                    {A B : Obj} (f : B  ⇒  A)
                    (β γ : NaturalTransformation Hom[ C ][-, A ] F) →
                  Setoid._≈_ (F₀ Nat[Hom[C][-,c],F] (F , A)) β γ →
                  Setoid._≈_ (F₀ F B) (η β B ⟨$⟩ f ∘ id) (F₁ F f ⟨$⟩ (η γ A ⟨$⟩ id))
         helper {F} {A} {B} f β γ β≈γ = S.begin
           η β B ⟨$⟩ f ∘ id          S.≈⟨ cong (η β B) (id-comm ∘ (⟺
           ↪  identityˡ)) ⟩
           η β B ⟨$⟩ id ∘ id ∘ f     S.≈⟨ commute β f CE.refl ⟩
           F₁ F f ⟨$⟩ (η β A ⟨$⟩ id) S.≈⟨ cong (F₁ F f) (β≈γ CE.refl) ⟩
           F₁ F f ⟨$⟩ (η γ A ⟨$⟩ id) S.□
           where module S where
                   open Setoid (F₀ F B) public
                   open SetoidR (F₀ F B) public
```

Here are a few downsides of not renaming:

1. The type of the function is difficult to comprehend; though it need not be.
   - ◇ Take `_≈₀_ = Setoid._≈_ (F₀ Nat[Hom[C][-,c],F] (F , A))`, and
   - ◇ Take `_≈₁_ = Setoid._≈_ (F₀ F B)`,
   - ◇ Then the type says: If $\beta \approx_0 \gamma$ then `η β B ⟨$⟩ f ∘ id ≈₁ F₁ F f ⟨$⟩ (η γ A ⟨$⟩ id)` —a naturality condition!
2. The short proof is difficult to read!
   - ◇ The repeated terms such as `η β B` and `η β A` could have been renamed with mnemoic-names such as $\eta_1$, $\eta_2$ or $\eta_s$, $\eta_t$ for 's'ource/1 and 't'arget/2.
   - ◇ Recall that functors `F` have projections $\mathtt{F}_i$, so the "mor"phism map on a given morphism `f` becomes `F₁ F f`, as in the excerpt above; however, using RATH-Agda's naming it would have been `mor F f`.

Since names are given a lower priority, one no longer needs to perform renaming. Instead, one is content with projections. The downside is now there are too many projections, leaving code difficult to comprehend. Moreover, this leads to inconsistent renaming.

## 3.3 Redundancy, Derived Features, and Feature Exclusion

A tenet of software development is not to over-engineer solutions. For example, if we need a notion of untyped composition, we may use `Monoid`. However, at a later stage, we may realise that units are inappropriate and so we need to drop them to obtain the weaker notion of

Semigroup —for instance, if we wish to model finite functions as hashmaps, we need to omit the identity functions since they may have infinite domains; and we cannot simply enforce a convention, say, to treat empty hashmaps as the identities since then we would lose the empty functions. In weaker languages, we could continue to use the monoid interface at the cost of "throwing an exception" whenever the identity is used. However, this breaks the *Interface Segregation Principle: Users should not be forced to bother with features they are not interested in* [**old-design-patterns-solid**]. A prototypical scenario is exposing an expressive interface, possibly with redundancies, to users, but providing a minimal self-contained counterpart by dropping some features for the sake of efficiency or to act as a "smart constructor" that takes the least amount of data to reconstruct the rich interface.

More concretely, in the Agda-categories library one finds concepts with expressive interfaces, with redundant features, prototypically named $\mathcal{X}$, along with their minimal self-contained versions, prototypically named $\mathcal{X}$Helper. In particular, the Category type and the natural isomorphism type are instances of such a pattern. The redundant features are there to make the lives of users easier; e.g., quoting Agda-categories, *We add a symmetric proof of associativity so that the opposite category of the opposite category is definitionally equal to the original category.* To underscore the intent, we present below a minimal setup needed to express the issue. The semigroup definition contains a redundant associativity axiom —which can be obtained from the first one by applying symmetry of equality. This is done purposefully so that the "opposite, or dual, transformer" _˘ is self-inverse on-the-nose; i.e., definitionally rather than propositionally equal. Definitionally equality does not need to be 'invoked', it is used silently when needed, thereby making the redundant setup worth it.

**Redundancy can lead to silently used equalities**

```
record Semigroup : Set₁ where
  constructor 𝒮
  field
    Carrier : Set
    _⨾_      : Carrier → Carrier → Carrier
    assocʳ : ∀ {x y z} →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)
    assocˡ : ∀ {x y z} →  x ⨾ (y ⨾ z)  ≡  (x ⨾ y) ⨾ z

    -- Notice:  assocˡ ≈ sym assocʳ

_˘ : Semigroup → Semigroup
(𝒮 Carrier _⨾_ assocʳ assocˡ) ˘  =  𝒮 Carrier (λ b a → a ⨾ b)  assocˡ assocʳ

˘˘≈id : ∀ {S} → (S ˘) ˘ ≡ S
˘˘≈id = refl
```

**On-the-nose Redundancy Design Pattern (Agda-Categories)**

Include redundant features if they allow certain common constructions to be definitionally equal, thereby requiring no overhead to use such an equality. Then, provide a smart constructor so users are not forced to produce the redundant features manually.

Incidentally, since this is not a library method, inconsistencies are bound to arise; in particular, in the $\mathcal{X}$ and $\mathcal{X}$`Helper` naming scheme: The `NaturalIsomorphism` type has `NIHelper` as its minimised version, and the type of symmetric monoidal categories is oddly called `Symmetric′` with its helper named `Symmetric`. Such issues could be reduced, if not avoided, if library methods could have been used instead.

It is interesting to note that duality forming operators, such as `_˘` above, are a design pattern themselves. How? In the setting of algebraic structures, one picks an operation to have its arguments flipped, then systematically 'flips' all proof obligations via a user-provided symmetry operator. We shall return to this as a library method in a future section.

Another example of purposefully keeping redundant features is for the sake of efficiency.

> For division semi-allegories, even though right residuals, restricted residuals, and symmetric quotients all can be derived from left residuals, we still assume them all as primitive here, since this produces more readable goals, and also makes connecting to optimised implementations easier. —RATH-Agda section 15.13

For instance, the above semigroup type could have been augmented with an ordering if we view `_⨾_` as a meet-operation. Instead, we lift such a derived operation as a primitive field, in case the user has a better implementation.

Simulating Default Implementations with Smart Constructors

```
record Order (S : Semigroup) : Set₁ where
  open Semigroup S public
  field
    _⊑_    : Carrier → Carrier → Set
    ⊑-def  : ∀ {x y} → (x ⊑ y) ≡ (x ⨾ y ≡ x)

  {- Results about _⨾_ and _⊑_ here ... -}

defaultOrder : ∀ S → Order S
defaultOrder S = let open Semigroup S
                 in record { _⊑_ = λ x y → x ⨾ y ≡ x ; ⊑-def = refl }
```

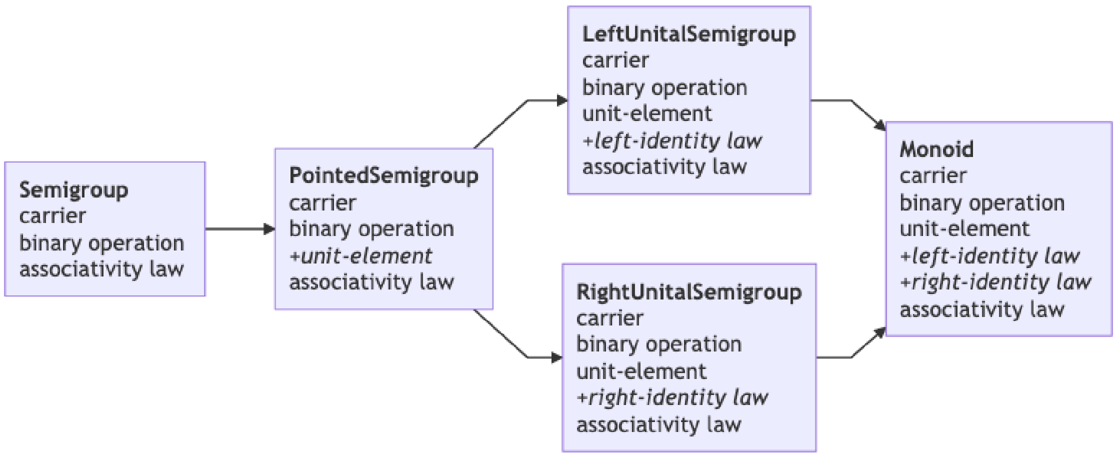
**Efficient Redundancy Design Pattern (RATH-Agda section 17.1)**

To enable efficient implementations, replace derived operators with additional fields for them and for the equalities that would otherwise be used as their definitions. Then, provide instances of these fields as derived operators, so that in the absence of more efficient implementations, these default implementations can be used with negligible penalty over a development that defines these operators as derived in the first place.

## 3.4 Extensions

In our previous discussion, we needed to drop features from `Monoid` to get `Semigroup`. However, excluding the unit-element from the monoid also required excluding the identity laws. More generally, all features reachable, via occurrence relationships, must be dropped when a particular feature is dropped. In some sense, a generated graph of features needs to be "ripped out" from the starting type, and the generated graph may be the whole type. As such, in general, we do not know if the resulting type even has any features.

Instead, in an ideal world, it is preferable to begin with a minimal interface then *extend* it with features as necessary. E.g., begin with `Semigroup` then add orthogonal features until `Monoid` is reached. Extensions are also known as *subclassing* or *inheritance*.



The libraries mentioned thus far generally implement extensions in this way. By way of example, here is how monoids could be built directly from semigroups along a particular path in the above hierarchy.

Extending Semigroup to Obtain Monoid

```
record Semigroup : Set₁ where
  field
    Carrier : Set
    _⨾_     : Carrier → Carrier → Carrier
    assoc  : ∀ {x y z} →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)

record PointedSemigroup : Set₁ where
  field semigroup : Semigroup
  open  Semigroup semigroup public {- (⋆) -}
  field Id : Carrier

record LeftUnitalSemigroup : Set₁ where
  field pointedSemigroup : PointedSemigroup
  open  PointedSemigroup pointedSemigroup public {- (⋆) -}
  field leftId : ∀ {x} → Id ⨾ x ≡ x

record Monoid : Set₁ where
  field leftUnitalSemigroup : LeftUnitalSemigroup
  open LeftUnitalSemigroup leftUnitalSemigroup public {- (⋆) -}
  field rightId : ∀ {x} → x ⨾ Id ≡ x

open Monoid  {- (⋆, *) -}

neato : ∀ {M} → Carrier M → Carrier M → Carrier M
neato {M} = _⨾_ M    {- (*); Possible due to all of the (⋆) above -}
```

**Extension Design Pattern**

To extend a structure $\mathcal{X}$ by new features `f`$_0$`, ..., f`$_n$ which may mention features of $\mathcal{X}$, make a new structure $\mathcal{Y}$ with fields for $\mathcal{X}$, `f`$_0$`, ..., f`$_n$. Then publicly open $\mathcal{X}$ in this new structure —see `(⋆)` above— so that the features of $\mathcal{X}$ are visible directly from $\mathcal{Y}$ to all users —see lines marked `(*)` above.

Notice how we accessed the binary operation `_⨾_` feature from `Semigroup` as if it were a native feature of `Monoid`. Unfortunately, `_⨾_` is only **superficially native** to `Monoid` —any actual instance, such as `woah` below, needs to define the binary operation in a `Semigroup` instance first, which lives in a `PointedSemigroup` instance, which lives in a `LeftUnitalSemigroup` instance.

Extensions are not flattened inheritance

```
woah : Monoid
woah = record { leftUnitalSemigroup
                 = record { pointedSemigroup
                             = record { semigroup = record { Carrier = {!!}
                                                            ; _⨾_     = {!!}
                                                            ; assoc   = {!!}
                                                            } -- Nesting level 3
                                      ; Id = {!!}
                                      } -- Nesting level 2
                          ; leftId = {!!}
                          } -- Nesting level 1
              ; rightId = {!!}
              }  -- Nesting level 0
```

This nesting scenario happens rather often, in one guise or another. The amount of syntactic noise required to produce a simple instantiation is unreasonable: **One should not be forced to work through the hierarchy if it provides no immediate benefit.**

Even worse, pragmatically speaking, to access a field deep down in a nested structure results in overtly lengthy and verbose names; as shown below. Indeed, in the above example, the monoid operation lives at the top-most level, we would need to access all the intermediary levels to simply refer to it. Such verbose invocations would immediately give way to helper functions to refer to fields lower in the hierarchy; yet another opportunity for boilerplate to leak in.

Extensions are not flattened inheritance

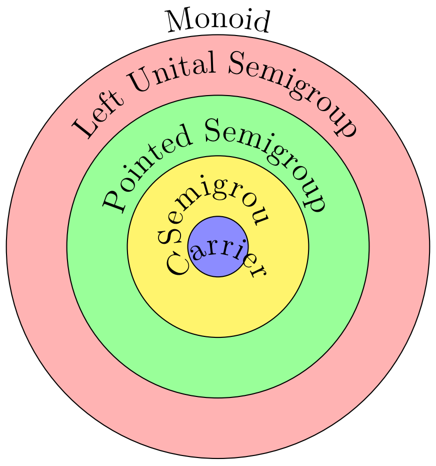
```
{- Without the (⋆) “public” declarations, projections are difficult! -}
carrier : Monoid → Set
carrier M = Semigroup.Carrier
              (PointedSemigroup.semigroup
                (LeftUnitalSemigroup.pointedSemigroup
                  (Monoid.leftUnitalSemigroup M)))
```

Instead of the above ‘staircase’, the following figure presents an alternative view of how extensions require deep projections.

While library designers may be content to build `Monoid` out of `Semigroup`, users should not be forced to learn about how the hierarchy was built. Even worse, when the library designers decide to incorporate, say, `RightUnitalSemigroup` instead of the left unital form, then all users’ code would break. Instead, it would be preferable to have a ‘flattened’ presentation for the users that “does not leak out implementation details”. We shall return to this in a future section.

It is interesting to note that diamond hierarchies cannot be trivially eliminated when providing fine-grained hierarchies. As such, we make no rash decisions regarding limiting

Figure 3.3: Projecting several levels down to get to `Carrier`



them —and completely forgoe the unreasonable possibility of forbidding them.

A more common example from programming is that of providing monad instances in Haskell. Most often users want to avoid tedious case analysis or prefer a sequential-style approach to producing programs, so they want to furnish a type constructor with a monad instance in order to utilise Haskell's `do`-notation. Unfortunately, this requires an applicative instances, which in turn requires a functor instance. However, providing the return-and-bind interface for monads allows us to obtain functor and applicative instances. Consequently, many users simply provide local names for the return-and-bind interface then use that to provide the default implementations for the other interfaces. In this scenario, **the standard approach is side-stepped** by manually carrying out a mechanical and tedious set of steps that not only wastes time but obscures the generic process and could be error-prone.

Instead, it would be desirable to 'flatten' the hierarchy into a single package, consisting of the fields throughout the hierarchy, possibly with default implementations, yet still be able to view the resulting package at base levels in the hierarchy —c.f., section 3.3. Another benefit of this approach is that it allows users to utilise the package without consideration of how the hierarchy was formed, thereby providing library designers with the freedom to alter it in the future.

## 3.5 Conclusion

After 'library spelunking', we are now in a position to summarise the problems encountered, when using existing[8] modules systems, that need a solution. From our learned lessons, we can then pinpoint a necessary feature of an ideal module system for dependently-typed languages.

[8]A comparison of module systems of other dependently-typed languages is covered in section ??.

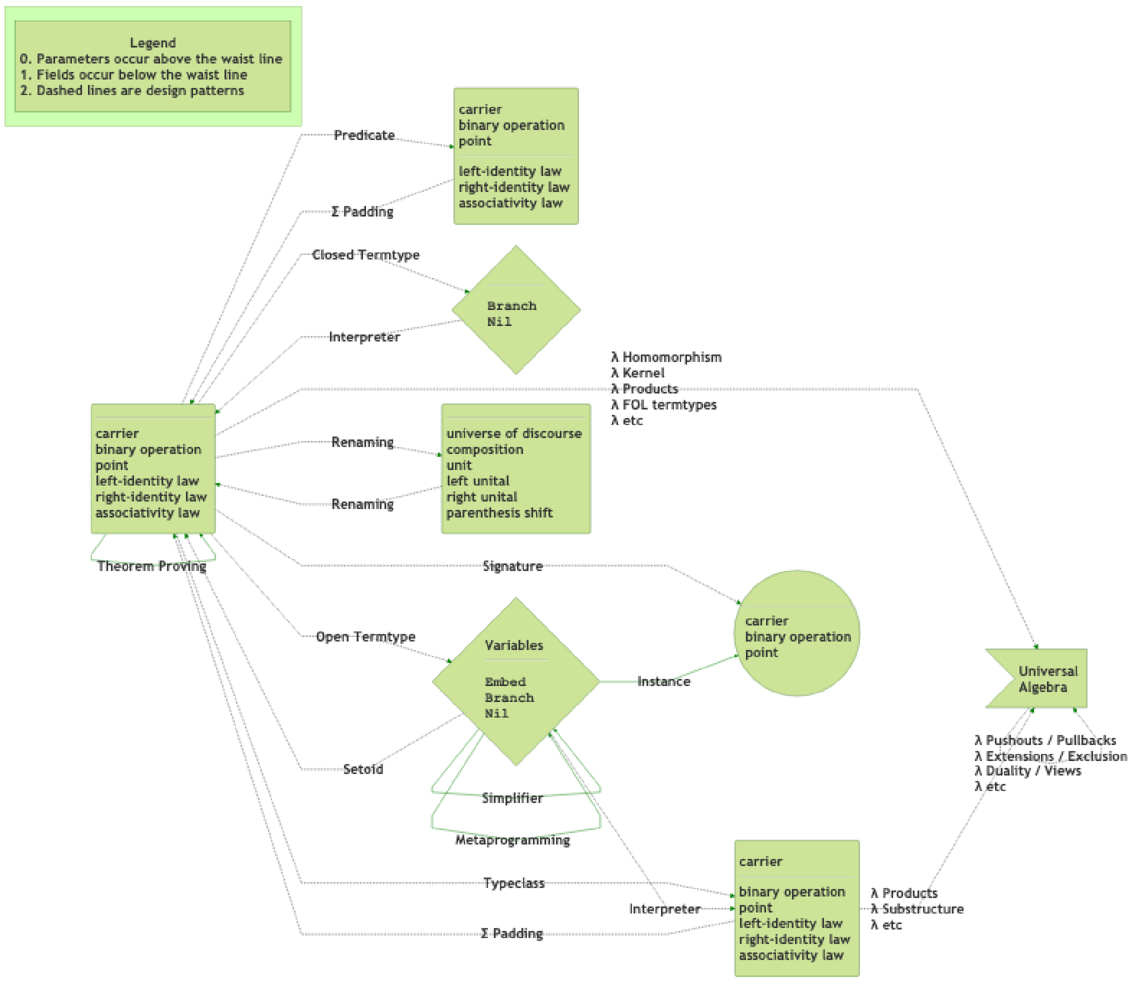
### 3.5.1 Lessons Learned

Systems tend to come with a pre-defined set of operations for built-in constructs; the user is left to utilise third-party pre-processing tools, for example, to provide extra-linguistic support for common repetitive scenarios they encounter.

More concretely, a large number of proofs can be discharged by merely pattern matching on variables —this works since the case analysis reduces the proof goal into a trivial reflexitivity obligation, for example. The number of cases can quickly grow thereby taking up space, which is unfortunate since the proof has very little to offer besides verifying the claim. In such cases, a pre-process, perhaps an "editor tactic", could be utilised to produce the proof in an auxiliary file, and reference it in the current file.

Perhaps more common is the renaming of package contents, by hand. For example, when a notion of preorder is defined with relation named `_≤_`, one may rename it and all references to it by, say, `_⊑_`. Again, a pre-processor or editor-tactic could be utilised, but many simply perform the re-write by hand —which is tedious, error prone, and obscures the generic rewriting method.

It would be desirable to **allow packages to be treated as first-class concepts that could be acted upon, in order to avoid third-party tools that obscure generic operations and leave them out of reach for the powerful typechecker of a dependently typed system.** Below is a summary of the design patterns mentioned above, using monoids as the prototypical structure. Some patterns we did not cover, as they will be covered in future sections.



Legend
0. Parameters occur above the waist line
1. Fields occur below the waist line
2. Dashed lines are design patterns
carrier
binary operation
point
left-identity law
right-identity law
associativity law
Predicate
Σ Padding
Closed Termtype
Interpreter
Branch
Nil
λ Homomorphism
λ Kernel
λ Products
λ FOL termtypes
λ etc
Renaming
Renaming
universe of discourse
composition
unit
left unital
right unital
parenthesis shift
Theorem Proving
Signature
Open Termtype
Variables
Embed
Branch
Nil
Instance
Setoid
Simplifier
Metaprogramming
Universal Algebra
λ Pushouts / Pullbacks
λ Extensions / Exclusion
λ Duality / Views
λ etc
Typeclass
Interpreter
Σ Padding
λ Products
λ Substructure
λ etc

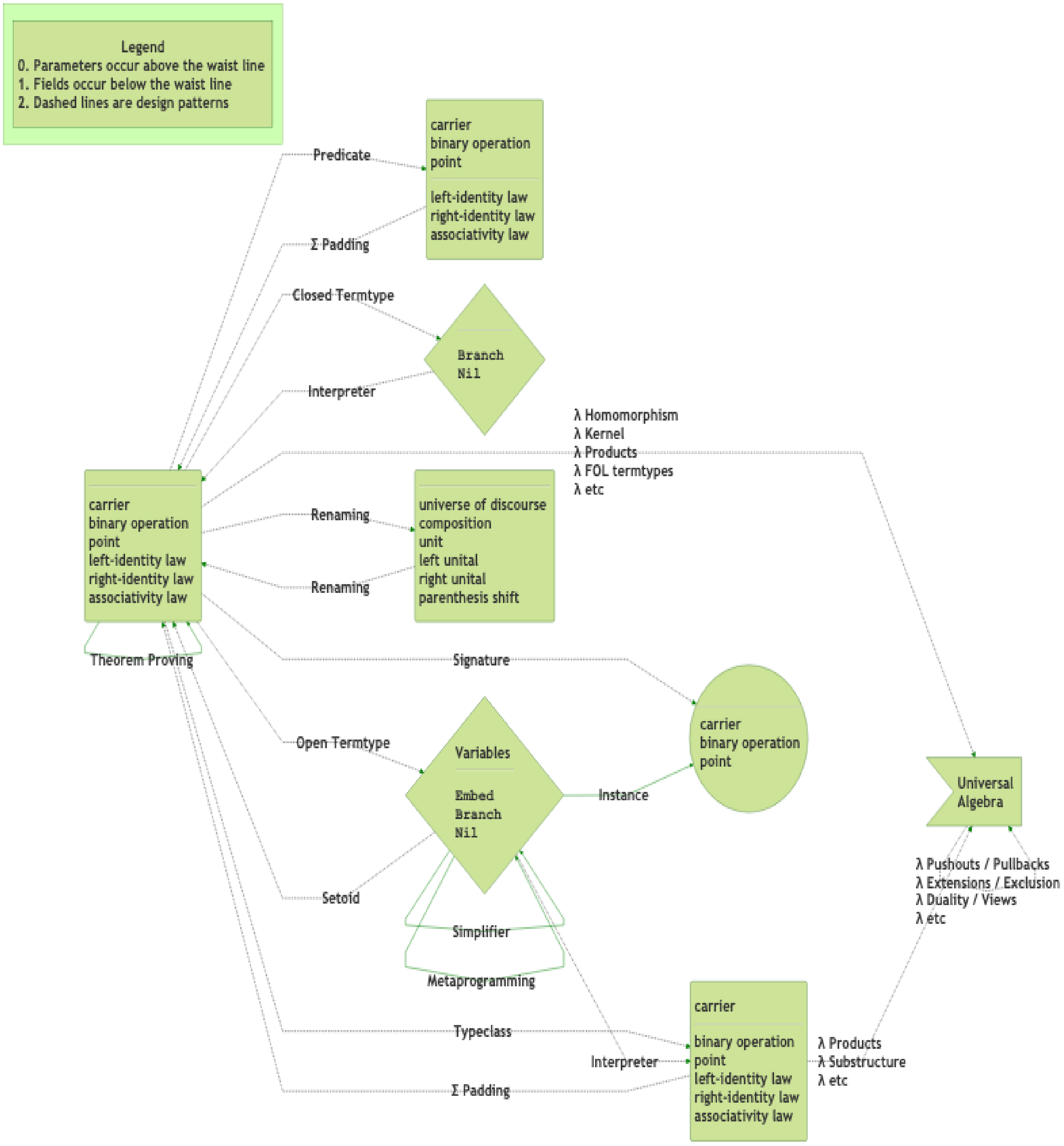


Figure 3.4: PL Research is about getting free stuff: From the left-most node, we can get a lot!

Remarks:

1. It is important to note that the `termtype` constructions could also be co-inductive, thereby yielding possibly infinitely branching syntax-trees.

   ◇ In the "simplify" pattern, one could use axioms as rewrite rules.

2. It is more convenient to restrict a carrier or to form products along carriers using the typeclass version.

3. As discussed earlier, the name *typeclass* is justified not only by the fact that this is the shape used by typeclasses in Haskell and Coq, but also that instance search for such records is supported in Agda by using the `instance` keyword.

There are many more design patterns in dependently-typed programming. Since grouping mechanisms are our topic, we have only presented those involving organising data.

### 3.5.2 One-Item Checklist for a Candidate Solution

An adequate module system for dependently-typed languages should make use of dependent-types as much as possible. As such, there is essentially one and only one primary goal for a module system to be considered reasonable for dependently-typed languages: Needless distinctions should be eliminated as much as possible.

The "write once, instantiate many" attitude is well-promoted in functional communities predominately for *functions*, but we will take this approach to modules as well, beyond the features of, e.g., SML functors. With one package declaration, one should be able to mechanically derive data, record, typeclass, product, sum formulations, among many others. All operations on the generic package then should also apply to the particular package instantiations.

This one goal for a reasonable solution has a number of important and difficult subgoals. The resulting system should be well-defined with a coherent semantic underpinning —possibly being a conservative extension—; it should support the elementary uses of pedestrian module systems; the algorithms utilised need to be proven correct with a mechanical proof assistant, considerations for efficiency cannot be dismissed if the system is to be usable; the interface for modules should be as minimal as possible, and, finally, a large number of existing use-cases must be rendered tersely using the resulting system without jeopardising runtime performance in order to demonstrate its success.

# 4 The `PackageFormer` Prototype

From the lessons learned from spelunking in a few libraries, we concluded that metaprogramming is an inescapable road on the journey toward first-class modules in DTLs. As such, we begin by forming an 'editor extension' to Agda with an eye toward the minimal number of 'meta-primitives' for forming combinators on modules. The extension is written in Lisp, an excellent language for rapid prototyping. The purpose of writing the editor extension is to show that the 'flattening' of value terms and module terms is not only feasible, but practical. The resulting tool resolves many of the issues discussed in section 3.

For the interested reader, the full implementation is presented *literately* as a discussion at `https://alhassy.github.io/next-700-module-systems/prototype/package-former.html`. We will not be discussing any Lisp code in particular.

Chapter Contents

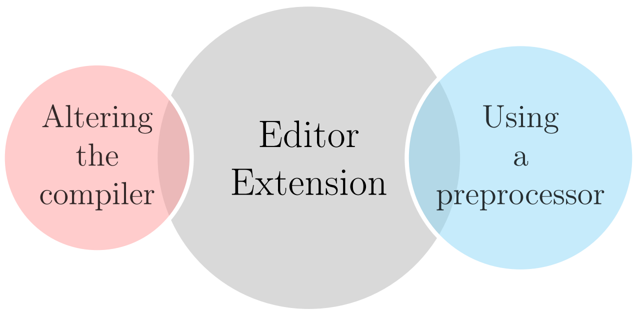
## 4.1 Why an editor extension? Why Lisp is reasonable?

At first glance, it is humorous[1] that a module extension for a statically dependently-typed language is written in a dynamically type checked language. However, *a lack of static types means some design decisions can be deferred as much as possible.*

**Why an editor extension?** Unless a language provides an extension mechanism, one is forced to either alter the language's compiler or to use a preprocessing tool —neither is particular appealing. The former is *dangerous*; e.g., altering the grammar of a language requires non-trivial propagated changes throughout its codebase, but even worse, it could lead to existing language features to suddenly break due to incompatibility with the added features. The latter is *tiresome*: It can be a nuisance to remember always invoke a preprocessor before compilation or type-checking, and it becomes extra baggage to future users of the codebase —i.e., a further addition to the toolchain that requires regular maintenance in order to be kept up to date with the core language. A middle-road between the two is not

[1]None of my colleagues thought Lisp was at all the 'right' choice; of-course, none of them had the privilege to use the language enough to appreciate it for the wonder that it is.

Figure 4.1: A reasonable middle path to growing a language

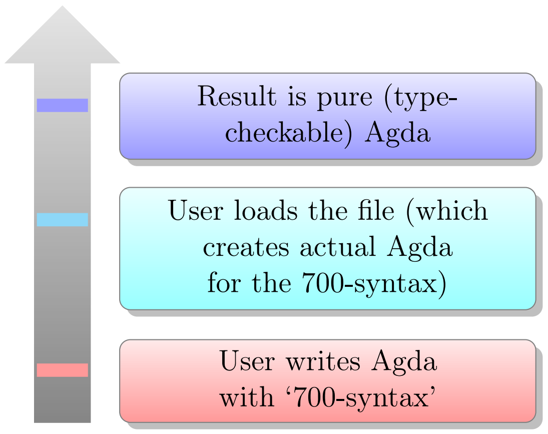


always possible. However, if the language's community subscribes to **one** Interactive Development Environment (IDE), then a **reasonable** approach to extending a language would be to *plug-in* the necessary preprocessing —to transform the extended language into the pure core language— in a saliently **silent** fashion such that users need not invoke it manually. Moreover, to mitigate the burden of increasing the toolchain, the salient preprocessing would **not transform user code** but instead **produce auxiliary files** containing core language code which are then *imported* by user code —furthermore, such import clauses could be automatically inserted when necessary. The benefit here is that **library users** need not know about the extended language features; since all files are in the core language with extended language feature appearing in special comments. Details can be found in section 4.2, while Figure **??** provides a bird's eye view.

**Why Emacs?** Agda code is predominately written in Emacs, so a practical and pragmatic editor extension would need be in Agda's de-facto IDE.

**Why Lisp?** Emacs is extensible using Elisp —a combination of a large porition of Common Lisp and a editor language supporting, e.g., buffers, text elements, windows, fonts— wherein literally every key may be remapped and existing utilities could easily be altered *without* having to recompile Emacs. In some sense, Emacs is a Lisp interpreter and state machine. This means, we can hook our editor extension **seamlessly into the existing Agda interface** and even provide tooltips, among other features, to quickly see what our extended Agda syntax transpiles into. Moreover, being a self-documenting editor, whenever a user of our tool wishes to see the documentation of a module combinator that they have written, or to read its Lisp elaboration, they merely need to invoke Emacs' help system —e.g., `C-h o` or `M-x describe-symbol`.

Figure 4.2: All stages transpire in *one* user-written file


Result is pure (type-checkable) Agda
User loads the file (which creates actual Agda for the 700-syntax)
User writes Agda with '700-syntax'

**Why textual transformations?** Metaprogramming is notoriously difficult to work with in typed settings, which mostly provide an opaque `Term` type thereby essentially resolving to working with untyped syntax trees. For instance, consider the Lisp term

```
(--map (+ it 2) '(1 2 3))
```

which may be written in Haskell as

```
map (λ it → it + 2) [1, 2, 3]
```

What is the type of `--map`? It expects a list after a functional expression whose bound variable is named `it`. Anaphoric macros like `--map` are thus not typeable as functions, but could be thought of as **new quantifiers**, implicitly binding the variable `it` in the first argument —in Haskell, one sees

```
map (λ it → ···) xs = [··· | it ← xs]
```

thereby cementing `map` as a form of variable binder. Thus, rather than work with abstract syntax terms for Agda, which requires non-trivial design decisions, we instead resolve to *rewrite* Agda phrases from an extended Agda syntax to legitimate existing syntax.

Finally, Lisp has a minimal number of built-in constructs which serve to define the usual host of expected language conveniences. That is, it provides an orthogonal set of 'meta-primitives' from which one may construct the 'primitives' used in day-to-day activities. E.g., with macro and lambda meta-primitives, one obtains the `defun` primitive for defining top-level functions. With Lisp as the implementing language, we were **implicitly encouraged** to seek meta-primitives for making modules.

## 4.2 Aim: *Scrap the Repetition*

Programming Language research is summarised, in essence, by the question: *If $\mathcal{X}$ is written manually, what information $\mathcal{Y}$ can be derived for free?* Perhaps the most popular instance is *type inference*: From the syntactic structure of an expression, its type can be derived. From a context, the `PackageFormer` editor extension can generate the many common design patterns discussed earlier in section 3.5.1; such as unbundled variations of any number wherein fields are exposed as parameters at the type level, term types for syntactic manipulation, arbitrary renaming, extracting signatures, and forming homomorphism types. In this section we discuss how `PackageFormer` works and provide a 'real-world' use case, along with a discussion.

The `PackageFormer` tool is an Emacs editor extension written in Lisp that is integrated seemlessly into the Agda Emacs interface: Whenver a user loads a file `X.agda` for interactive typechecking, with the usual Agda keybinding `C-c C-l`, `PackageFormer` performs the following steps:

1. Parse any comments `{-700 ··· -}` containing fictitious Agda code,
2. Produce legitimate Agda code for the '700-comments' into a file `X_generated.agda`,
3. Add to `X.agda` a call to import `X_generated.agda`, if need be; and, finally,
4. Actually perform the expected typechecking.
   - ◇ For every 700-comment declaration $\mathcal{L}$ = $\mathcal{R}$ in the source file, the name $\mathcal{L}$ obtains a tooltip which mentions its specification $\mathcal{R}$ and the resulting legitimate Agda code. This feature is indispensable as it lets one generate grouping mechanisms and quickly ensure that they are what one intends them to be.

Here is an example of contents in a 700-comment. The first eight lines, starting at line 1, are essentially an Agda `record` declaration but the `field` qualifier is absent. The declaration is intended to name an abstract context, a sequence of "name : type" pairs as discussed at length in chapter 2, but we use the name `PackageFormer` instead of 'context, signature, telescope', nor 'theory' since those names have existing biased connotations —besides, the new name is more 'programmer friendly'.

M-Sets are sets 'Scalar' acting '_·_' on semigroups 'Vector'

```
1  PackageFormer M-Set : Set₁ where
2     Scalar  : Set
3     Vector  : Set
4     _·_     : Scalar → Vector → Vector
5     𝟙       : Scalar
6     _×_     : Scalar → Scalar → Scalar
7     leftId  : {v : Vector}  →  𝟙 · v  ≡  v
8     assoc   : {a b : Scalar} {v : Vector} → (a × b) · v  ≡  a · (b · v)
```

**Aside:** The names have been chosen to stay relatively close to the real-world examples presented in chapter 3. ( The name **M-Set** comes from *monoid acting on a set*; in our example, `Scalar` values may act on `Vector` values to produce new `Scalar` values. ) The programmer may very well appreciate this example if the names `Scalar`, `𝟙`, `_×_`, `Vector`, `_·_` were chosen to be `Program`, `do-nothing`, `_⨾_`, `Input`, `run`. With this new naming, `leftId` says *running the empty program on any input, leaves the input unchanged*, whereas `assoc` says *to run a sequence of programs on an input, the input must be threaded through the programs*. Whence, **M-Sets abstract program execution**.

Different Ways to Organise (“interpret”

```
-- M-Sets as records, possibly with renaming, or with parameters
Semantics           = M-Set ⟴ record
Semantics𝒟           = Semantics ⟴ rename (λ x → (concat x "𝒟"))
Semantics₃          =  Semantics :waist 3

-- Duality; chaning the order of the action (c.f., “run” above)
Left-M-Set          = M-Set ⟴ record
Right-M-Set         = Left-M-Set ⟴ flipping "_·_" :renaming "leftId to rightId"

-- Keeping only the ‘syntactic interface’, say, for serialisation or automation
ScalarSyntax        = M-Set ⟴ primed ⟴ data "Scalar′"
Signature           = M-Set ⟴ record ⟴ signature
Sorts               = M-Set ⟴ record ⟴ sorts

-- Collapsing different features to obtain the notion of “monoid”
𝒱-one-carrier        = renaming "Scalar to Carrier; Vector to Carrier"
𝒱-compositional      = renaming "_×_ to _⨾_; _·_ to _⨾_"
𝒱-monoidal           = one-carrier ⟴ compositional ⟴ record

-- Obtaining parts of the monoid hierarchy (see chapter 3) from M-Sets
LeftUnitalSemigroup = M-Set ⟴ monoidal
Semigroup           = M-Set ⟴ keeping "assoc" ⟴ monoidal
Magma               = M-Set ⟴ keeping "_×_" ⟴ monoidal
```

These manually written ∼25 lines elaborate into the ∼100 lines of raw, legitimate, Agda syntax below —line breaks are denoted by the symbol ‘↪’ rather than inserted manually, since all subsequent code snippets in this section are **entirely generated** by `PackageFormer`. The result is nearly a **400% increase in size**; that is, our fictitious code will save us a lot of repetition.

`PackageFormer` module combinators are called *variationals* since they provide a variation on an existing grouping mechanism. The syntax `p` ⟴ $v_1$ ⟴ ⋯ ⟴ $v_n$ is tantamount to explicit forward function application $v_n$ ($v_{n-1}$ (⋯ ($v_1$ `p`))). With this understanding, we can explain the different ways to organise M-sets.

**Line 1** The context of `M-Set`s is declared.

This is the traditional Agda syntax " `record M-Set : Set₁ where` " except the we use the word `PackageFormer` to avoid confusion with the existing record concept, but we also *omit* the need for a `field` keyword and *forbid* the existence of parameters.

**Conflating fields, parameters, and definitional extensions**

The lack of a `field` keyword and forbidding parameters means that arbitrary programs may 'live within' a `PackageFormer` and it is up to a variational to decide how to treat them and their optional definitions.

Such abstract contexts have no concrete form in Agda and so no code is generated.

**Line 10** The `record` variational is invoked to transform the abstract context `M-Set` into a valid Agda record declaration, with the key word `field` inserted as necessary. Later, its first 3 fields are lifted as parameters using the meta-primitive `:waist`.

**Arbitrary functions act on modules**

When only one variational is applied to a context, the one and only '⟴' may be omitted. As such, $\texttt{Semantics}_3$ is defined as `Semantics rename f`, where `f` is the decoration function. In this form, one is tempted to believe

```
_rename_ : PackageFormer → (Name → Name) → PackageFormer
```

That is, we have a binary operation in which functions may act on modules —this is yet a new feature that Agda cannot perform.

Record / decorated renaming / typeclass forms

```
record Semantics : Set₁ where
    field Scalar                : Set
    field Vector                : Set
    field _·_           : Scalar → Vector → Vector
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar
    field leftId                : {v : Vector}  →  𝟙 · v  ≡  v
    field assoc         : {a b : Scalar} {v : Vector} → (a × b) · v  ≡  a ·
    ↪  (b · v)


{- Semantics𝒟          = Semantics ⟴ rename (λ x → (concat x "𝒟")) -}
record Semantics𝒟 : Set₁ where
    field Scalar𝒟               : Set
    field Vector𝒟               : Set
    field _·𝒟_          : Scalar𝒟 → Vector𝒟 → Vector𝒟
    field 𝟙𝒟            : Scalar𝒟
    field _×𝒟_          : Scalar𝒟 → Scalar𝒟 → Scalar𝒟
    field leftId𝒟               : {v : Vector𝒟}  →  𝟙𝒟 ·𝒟 v  ≡  v
    field assoc𝒟                : {a b : Scalar𝒟} {v : Vector𝒟} → (a ×𝒟 b)
    ↪  ·𝒟 v  ≡  a ·𝒟 (b ·𝒟 v)
    toSemantics         : let View X = X in View Semantics ;    toSemantics =
    ↪  record {Scalar = Scalar𝒟;Vector = Vector𝒟;_·_ = _·𝒟_;𝟙 = 𝟙𝒟;_×_ =
    ↪  _×𝒟_;leftId = leftId𝒟;assoc = assoc𝒟}


{- Semantics₃          =  Semantics :waist 3 -}
record Semantics₃ (Scalar : Set) (Vector : Set) (_·_ : Scalar → Vector →
↪  Vector) : Set₁ where
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar
    field leftId                : {v : Vector}  →  𝟙 · v  ≡  v
    field assoc         : {a b : Scalar} {v : Vector} → (a × b) · v  ≡  a ·
    ↪  (b · v)
```

Likewise, line 15, mentions another combinator `_flipping_ : PackageFormer` $\boxed{\rightarrow}$ `Name` $\boxed{\rightarrow}$ `PackageFormer`; however, it also takes an *optional keyword argument* `:renaming`, which simply renames the given pair. The notation of keyword arguments is inherited[2] from Lisp.

[2]More accurately, the '⟴'-based mini-language for variationals is realised as a Lisp macro and so, in general, the right side of a declaration in 700-comments is interpreted as valid Lisp modulo this mini-language: `PackageFormer` names and variationals are variables in the Emacs environment —for declaration purposes, and to avoid touching Emacs specific utilities, variationals `f` are actually named `𝒱-f`. One may quickly obtain the documentation of a variational `f` with `C-h o RET 𝒱-f` to see how it works.

Duality: Sets can act on semigroups from the left or the right

```
record Left-M-Set : Set₁ where
    field Scalar                : Set
    field Vector                : Set
    field _·_           : Scalar → Vector → Vector
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar
    field leftId                : {v : Vector}  →  𝟙 · v  ≡  v
    field assoc         : {a b : Scalar} {v : Vector} → (a × b) · v  ≡  a ·
    ↪  (b · v)


{- Right-M-Set          = Left-M-Set ⟴ flipping "_·_" :renaming "leftId to
↪  rightId" -}
record Right-M-Set : Set₁ where
    field Scalar                : Set
    field Vector                : Set
    field _·_           :  Vector  → Scalar  →  Vector
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar
    field rightId               : let _·_ = λ x y → _·_ y x in {v : Vector}
    ↪  →  𝟙 · v  ≡  v
    field assoc         : let _·_ = λ x y → _·_ y x in {a b : Scalar} {v :
    ↪  Vector} → (a × b) · v  ≡  a · (b · v)
    toLeft-M-Set                : let _·_ = λ x y → _·_ y x in let View X = X
    ↪  in View Left-M-Set ;      toLeft-M-Set  = let _·_ = λ x y → _·_ y x
    ↪  in   record {Scalar = Scalar;Vector = Vector;_·_ = _·_;𝟙 = 𝟙;_×_ =
    ↪  _×_;leftId = rightId;assoc = assoc}
```

Notice how `Semantics𝒟` was *built from* a concrete context, namely the `Semantics` record. As such, every instance of `Semantics𝒟` can be transformed as an instance of `Semantics`: This view —see Section **??**— is automatically generated and named `toSemantics` above, by default. Likewise, `Right-M-Set` was derived from `Left-M-Set` and so we have automatically have a view `Right-M-Set` $\boxed{\to}$ `Left-M-Set`.

It is important to remark that the mechanical construction of such views (coercions) is **not built-in**, but rather a *user-defined* variational that is constructed from `PackageFormer`'s meta-primitives.

**Line 19** An algebraic data type is a tagged union of symbols, terms, and so is one type —see section 2.4.4. We can view a context as such a termtype by declaring one sort of the context to act as the termtype and then keep only the function symbols that target it —this is the **core idea** that is used when we operate on Agda `Term`s in the next chapter. Furthermore, recall from Chapter 2, symbols that target `Set` are considered sorts and if we keep only the symbols targeting a sort, we have a signature. ( By allowing symbols to be of type `Set`, we actually have **generalised contexts**. )

Termtypes and lawless presentations

```
data ScalarSyntax : Set where
    𝟙′          : ScalarSyntax
    _×′_                : ScalarSyntax → ScalarSyntax → ScalarSyntax


{- Signature            = M-Set ⟴ record ⟴ signature -}
record Signature : Set₁ where
    field Scalar                  : Set
    field Vector                  : Set
    field _·_           : Scalar → Vector → Vector
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar


{- Sorts                = M-Set ⟴ record ⟴ sorts -}
record Sorts : Set₁ where
    field Scalar                  : Set
    field Vector                  : Set
```

( The priming decoration is needed so that the names `𝟙, _×_` do not pollute the global name space. )

**Line 24** Declarations starting with " $\mathcal{V}$- " indicate that a new variation is to be formed, rather than a new grouping mechanism. For instance, the user-defined `one-carrier` variational identifies both the `Scalar` and `Vector` sorts, whereas `compositional` identifies the binary operations; then, finally, `monoidal` performs both of those operations and also produces a concrete Agda `record` formulation.

User defined variationals are applied as if they were built-ins —interestingly, only `:waist` and `_⟴_` are built-in meta-primitives, the other primitives discussed thus far build upon less than 5 meta-primitives.

Conflating features gives familiar structures

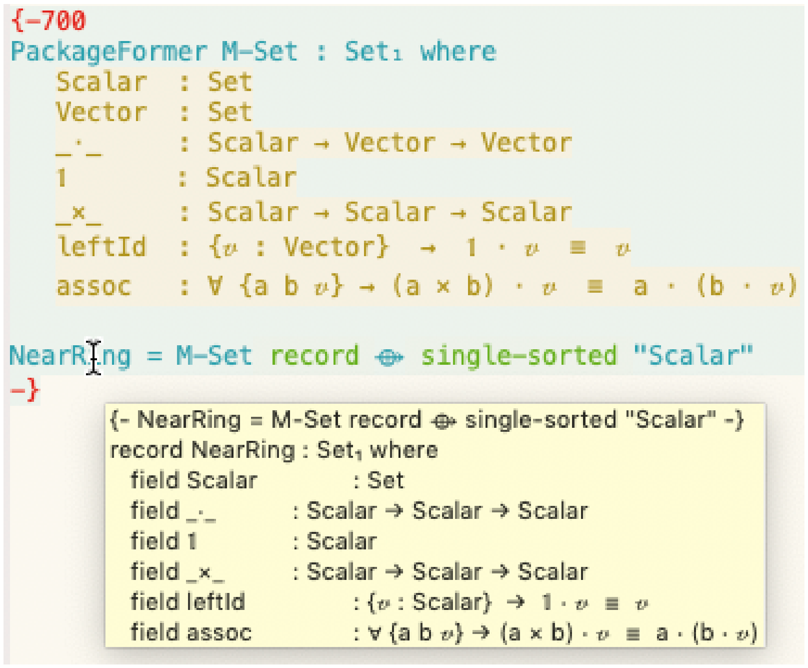
```
record LeftUnitalSemigroup : Set₁ where
    field Carrier               : Set
    field _⨾_          : Carrier → Carrier → Carrier
    field 𝟙             : Carrier
    field leftId                : {𝓋 : Carrier}  →  𝟙 ⨾ 𝓋  ≡  𝓋
    field assoc          : {a b : Carrier} {𝓋 : Carrier} → (a ⨾ b) ⨾ 𝓋  ≡  a ⨾
    ↪  (b ⨾ 𝓋)


{- Semigroup          = M-Set ⟴ keeping "assoc" ⟴ monoidal -}
record Semigroup : Set₁ where
    field Carrier               : Set
    field _⨾_          : Carrier → Carrier → Carrier
    field assoc          : {a b : Carrier} {𝓋 : Carrier} → (a ⨾ b) ⨾ 𝓋  ≡  a ⨾
    ↪  (b ⨾ 𝓋)


{- Magma              = M-Set ⟴ keeping "_×_" ⟴ monoidal -}
record Magma : Set₁ where
    field Carrier               : Set
    field _⨾_          : Carrier → Carrier → Carrier
```

As shown in Figure 4.3, the source file is furnished with tooltips displaying the 700-comment that a name is associated with, as well as the full elaboration into legitimate Agda syntax. In addition, the above generated elaborations also document the 700-comment that produced them. Moreover, since the editor extension results in valid code in an auxiliary file, future users of a library need not use the `PackageFormer` extension at all —thus we essentially have a static **editor tactic** similar to Agda's (Emacs interface) proof finder.

## 4.3 Practicality

Herein we demonstrate how to use this system from the perspective of *library designers*. That is to say, we will demonstrate how common desirable features encountered "in the wild" —chapter 3— can be used with our system. The exposition here follows section 2 of the *Theory Presentation Combinators* **tpc**, reiterating many the ideas therein. These features are **not built-in** but instead are constructed from a small set of meta-primitives, just as a small core set of language features give way to complex software programs. Moreover, user may combine the meta-primitives —using Lisp— to **extend** the system to produce grouping mechanisms for any desired purpose.

The few constructs demonstrated in this section not only create new grouping mechanisms from old ones, but also create maps from the new, child, presentations to the old parent presentations. For example, a theory extended by new declarations comes equipped with a



```
{-700
PackageFormer M-Set : Set₁ where
   Scalar  : Set
   Vector  : Set
   _·_     : Scalar → Vector → Vector
   1       : Scalar
   _×_     : Scalar → Scalar → Scalar
   leftId  : {𝓋 : Vector}  →  1 · 𝓋  ≡  𝓋
   assoc   : ∀ {a b 𝓋} → (a × b) · 𝓋  ≡  a · (b · 𝓋)

NearRing = M-Set record ⟴ single-sorted "Scalar"
-}
```

```
{- NearRing = M-Set record ⟴ single-sorted "Scalar" -}
record NearRing : Set₁ where
  field Scalar         : Set
  field _·_       : Scalar → Scalar → Scalar
  field 1         : Scalar
  field _×_       : Scalar → Scalar → Scalar
  field leftId         : {𝓋 : Scalar} → 1 · 𝓋 ≡ 𝓋
  field assoc          : ∀ {a b 𝓋} → (a × b) · 𝓋 ≡ a · (b · 𝓋)
```

Figure 4.3: Hovering to show details. Notice special syntax has default colouring: Red for PackageFormer delimiters, yellow for elements, and green for variationals.

map that forgets the new declarations to obtain an instance of the original theory. Such morphisms are tedious to write out, and our system provides them for free. The user can implement such features using our 5 meta-primitives —but we have implemented a few to show that the meta-primitives are deserving of their name.

**Do-it-yourself Extendability**

In order to make the editor extension immediately useful, and to substantiate the claim that **common module combinators can be defined using the system**, we have implemented a few notable ones, as described in the table below. The implementations, in the user manual, are discussed along with the associated Lisp code and use cases.

## Summary of Sample Variationals Provided With The System

| **Name** | **Description** |
|---|---|
| `record` | Reify a PackageFormer as a valid *Agda record* |
| `data` | Reify a PackageFormer as a valid Agda algebraic data type, $\mathcal{W}$-type |
| `extended-by` | Extend a PackageFormer by a string-";"-list of declaration |
| `union` | Union two PackageFormers into a new one, maintaining relationships |
| `flipping` | Dualise a binary operation or predicate |
| `unbundling` | Consider the first $N$ elements, which may have definitions, as parameters |
| `open` | Reify a given PackageFormer as a parameterised *Agda module* declaration |
| `opening` | Open a record as a module exposing only the given names |
| `open-with-decoration` | Open a record, exposing all elements, with a given decoration |
| `keeping` | Largest well-formed PackageFormer consisting of a given list of elements |
| `sorts` | Keep only the types declared in a grouping mechanism |
| `signature` | Keep only the elements that target a sort, drop all else |
| `rename` | Apply a `Name → Name` function to the elements of a PackageFormer |
| `renaming` | Rename elements using a list of "to"-separated pairs |
| `decorated` | Append all element names by a given string |
| `codecorated` | Prepend all element names by a given string |
| `primed` | Prime all element names |
| `subscripted`$_i$ | Append all element names by subscript `i : 0..9` |
| `hom` | Formulate the notion of homomorphism of parent PackageFormer algebras |

The following table has the **five meta-primitives** from which all variationals are borne, followed by two others that are useful for extending the system by making your own grouping mechanisms and operations on them. Using these requires a small amount of Lisp.

## Metaprogramming Meta-primitives for Making Modules

| **Name** | **Description** |
|---|---|
| `:waist` | Consider the first $N$ elements as, possibly ill-formed, parameters. |
| `:kind` | Valid Agda grouping mechanisms: `record, data, module`. |
| `:level` | The Agda level of a PackageFormer. |
| `:alter-elements` | Apply a `List Element → List Element` function over a PackageFormer. |
| ⟴ | Compose two variational clauses in left-to-right sequence. |
| `map` | Map a `Element → Element` function over a PackageFormer. |
| `generated` | Keep the sub-PackageFormer whose elements satisfy a given predicate. |

`PackageFormer` packages are an **implementation of the idea** of packages fleshed out in Chapter 2. Tersely put, a `PackageFormer` package is essentially a pair of tags —alterable by `:waist` to determine the height delimiting parameters from fields, and by `:kind` to determine a possible legitimate Agda representation that lives in a universe dictated by `:level`— as well as a list of declarations (elements) that can be manipulated with `:alter-elements`. Any variational $v$ that takes an argument of type $\tau$ can be thought of as a **binary packaged-valued operator**,

$$\_v\_ \ : \ \ \texttt{PackageFormer} \to \tau \to \texttt{PackageFormer}$$

With this perspective, the *sequencing variational combinator* '$\oplus\!\!\rightarrow$' is essentially forward function composition/application. Details can be found on the associated webpage; whereas the next chapter provides an Agda function-based semantics.

The remainder of this section is an exposition of notable **user-defined** combinators — i.e., those which can be constructed using the system's meta-primitives and a small amount of Lisp. Along the way, for each example, we show both the terse specfication using `PackageFormer` and its elaboration into pure typecheckable Agda. In particular, since packages are essentially a list of declarations —see Chapter 2— we begin in section 4.3.1 with the `extended-by` combinator which "grows a package". Then, in section 4.3.2, we show how *Agda users* can **quickly**, with a *tiny* amount of Lisp[3] knowledge, make useful variationals to abbreviate commonly occurring situations, such as a method to adjoin named operation properties to a a package. After looking at a `renaming` combinator, in section 4.3.3, and its properties that make it resonable; we show the Lisp code, in section 4.3.4 required for a pushout construction on packages. Of note is how Lisp's keyword argument feature allows the *verbose* 5-argument pushout operation to be **used** *easily* as a 2-argument operation, with other arguments optional. This construction is shown to generalise set union (disjoint and otherwise) and provide support for granular hierarchies thereby solving the so-called 'diamond problem'. Afterword, in section 4.3.5, we turn to another example of **formalising common patterns** —see Chapter 3— by showing how the idea of duality, not much used in simpler type systems, is used to mechanically produce new packages from old ones. Then, in section 4.3.6, we show how the interface segregation principle can be **applied after the fact**. Finally, we close in section 4.3.7 with a measure of the systems immediate practicality.

### 4.3.1 Extension

The simplest operation on packages is when one package is included, verbatim, in another. Concretely, consider `Monoid` —which consists of a number of *parameters* and the derived result `▯-unique`— and `CommutativeMonoid`$_0$ below.

[3]The `PackageFormer` manual provides the expected Lisp methods one is interested in, such as `(list x`$_0$ `...` `x`$_n$`)` to make a list and `first, rest` to decompose it, and `(--map (···it···) xs)` to traverse it. Moreover, an Emacs Lisp cheat sheet covering is provided.

Manually Repeating the entirety of ‘Monoid’ within ‘CommutativeMonoid$_0$’

```
{-700
PackageFormer Monoid : Set₁ where
   Carrier : Set
   _·_     : Carrier → Carrier → Carrier
   assoc   : {x y z : Carrier} → (x · y) · z  ≡  x · (y · z)
   ▯       : Carrier
   leftId  : {x : Carrier} → ▯ · x  ≡ x
   rightId : {x : Carrier} → x · ▯  ≡ x
   ▯-unique : ∀ {e} (lid : ∀ {x} → e · x ≡ x) (rid : ∀ {x} → x · e ≡ x) → e ≡ ▯
   ▯-unique lid rid = ≡.trans (≡.sym leftId) rid

PackageFormer CommutativeMonoid₀ : Set₁ where
   Carrier : Set
   _·_     : Carrier → Carrier → Carrier
   assoc   : {x y z : Carrier} → (x · y) · z  ≡  x · (y · z)
   ▯       : Carrier
   leftId  : {x : Carrier} →  ▯ · x  ≡ x
   rightId : {x : Carrier} →  x · ▯  ≡ x
   comm    : {x y : Carrier} →  x · y  ≡  y · x
   ▯-unique : ∀ {e} (lid : ∀ {x} → e · x ≡ x) (rid : ∀ {x} → x · e ≡ x) → e ≡ ▯
   ▯-unique lid rid = ≡.trans (≡.sym leftId) rid
-}
```

As expected, the only difference is that `CommutativeMonoid₀` adds a `comm`utatity axiom. Thus, given `Monoid`, it would be **more economical** to define:

Economically declaring only the new additions to ‘Monoid’

```
{-700
CommutativeMonoid = Monoid extended-by "comm : {x y : Carrier} →  x · y  ≡  y · x"
-}
```

As discussed in the previous section, mouse-hovering over the left-hand-side of this declaration gives a tooltip showing the resulting elaboration, which is identical to `CommutativeMonoid₀` above along with a forgetful operation, shown below. The tooltip shows the *expanded* version of the theory, which is **what we want to specify but not what we want to enter manually**. As discussed in section 3.4, to obtain this specification of `CommutativeMonoid` in the current implementation of Agda, one would likely declare a record with two fields —one being a `Monoid` and the other being the commutativity constraint— however, this only gives the appearance of the above specification for consumers; those who produce instances of `CommutativeMonoid` are then forced to know the particular hierarchy and must provide a `Monoid` value first. It is a happy coincidence that our system alleviates such an issue; i.e., we have **flattened extensions**.

Alternatively, we may reify the new syntactical items as concrete Agda supported `record`s as follows.

Every ‘CommutativeMonoid’ is automatically viewable as a ‘Monoid’

```
{-700
MonoidR             = Monoid -⊕→ record
CommutativeMonoidR = MonoidR extended-by "comm : {x y : Carrier} →  x · y  ≡  y ·
↪  x" -⊕→ record
-}

neato : CommutativeMonoidR → MonoidR
neato = CommutativeMonoidR.toMonoidR
```

Transport

It is important to notice that the *derived* result `𝕀-unique`, while proven in the setting of `Monoid`, is not only available via the morphism `toMonoidR` but is also available directly since it is also a member of `CommutativeMonoidR`.

One may use the call `P = Q extended-by R :adjoin-retract nil` to extend `Q` by declaration `R` but avoid having a view (coercion) `P` $\boxed{\rightarrow}$ `Q`. Of-course, `extended-by` is *user-defined* and we have simply chosen to adjoint retract views by default; the online documentation shows how users can define their own variationals.

### 4.3.2 Defining a Concept Only Once

From a library-designer’s perspective, our definition of `CommutativeMonoid` has the commutativity property ‘hard coded’ into it. If we wish to speak of commutative magmas —types with a single commutative operation— we need to hard-code the property once again. If, at a later time, we wish to move from having arguments be implicit to being explicit then we need to track down every hard-coded instance of the property then alter them —having them in-sync becomes an issue.

Instead, the system lets us ‘build upon’ the `extended-by` combinator: We make an associative list of names and properties, then string-replace the meta-names *op, op′, rel* with the provided user names. The definition below uses functional methods and should not be inaccessible to Agda programmers[4].

[4]The method call `(s-replace old new s)` replaces all occurrences of string `old` by `new` in the given string `s` ; whereas (`pcase e` ($x_0$ $y_0$) $\boxed{\dots}$ ($x_n$ $y_n$)) pattern matches on `e` and performs the first $y_i$ if `e` = $x_i$, otherwise it returns `nil`.

The ‘postulating’ variational

```
(𝒱 postulating bop prop (using bop) (adjoin-retract t)
 = "Adjoin a property PROP for a given binary operation BOP.

   PROP may be a string: associative, commutative, idempotent, etc.

   Some properties require another operator or a relation; which may
   be provided via USING.

   ADJOIN-RETRACT is the optional name of the resulting retract morphism.
   Provide nil if you do not want the morphism adjoined.

   With this variational, a definition is only written once.
   "
   extended-by
    (s-replace "op" bop (s-replace "rel" using (s-replace "op′" using
     (pcase prop
      ("associative"   "assoc : ∀ x y z → op (op x y) z ≡ op x (op y z)")
      ("commutative"   "comm  : ∀ x y   → op x y ≡ op y x")
      ("idempotent"    "idemp : ∀ x     → op x x ≡ x")
      ("left-unit"     "unitˡ : ∀ x y z → op e x ≡ e")
      ("right-unit"    "unitʳ : ∀ x y z → op x e ≡ e")
      ("absorptive"    "absorp  : ∀ x y  → op x (op′ x y) ≡ x")
      ("reflexive"     "refl    : ∀ x y  → rel x x")
      ("transitive"    "trans   : ∀ x y z → rel x y → rel y z → rel x z")
      ("antisymmetric" "antisym : ∀ x y → rel x y → rel y x → x ≡ z")
      (_ (error "𝒱-postulating does not know the property “%s”" prop))
      )))) :adjoin-retract 'adjoin-retract)
```

Lisp Syntax

The syntax of variational declarations was discussed in the previous section; one has access to the entirety of Emacs Lisp when forming such definitions. In particular, notice that their is a *documentation string* for the variational `postulating` so that when a user mouse-hovers over any occurrence of it within an Agda file, the documentation string appears as a tooltip. The first line declares the variational `postulating` to take two explicit arguments `bop, prop` followed by two optional arguments `:using, :adjoin-retract` that have default values `bop, t`.

This variational simply looks up the requested property `prop` in its local (hard coded) database, rewrites the *prototypical* name *op* with the given `bop`, then extends the given package with this property by calling on the `extended-by` variational. In Lisp, call sites for optional keyword arguments require a prefix colon; e.g., the last line of the above definition invokes `extended-by` and simply propagates the request to either adjoin, or not, a retract to the parent package.

We can extend this database of properties as needed with relative ease. Here is an example use along with its elaboration.

Example Use

```
{-700
PackageFormer Magma : Set₁ where
  Carrier : Set
  _·_      : Carrier → Carrier → Carrier

RawRelationalMagma = Magma extended-by "_≈_ : Carrier → Carrier → Set" ⟴ record

RelationalMagma    = RawRelationalMagma postulating "_·_" "congruence" :using "_≈_"
↪  ⟴ record
-}
```

Associated Elaboration

```
record RawRelationalMagma : Set₁ where
    field Carrier       : Set
    field op        : Carrier → Carrier → Carrier
    toType      : let View X = X in View Type ; toType = record {Carrier = Carrier}
    field _≈_       : Carrier → Carrier → Set
    toMagma     : let View X = X in View Magma ;    toMagma = record {Carrier =
    ↪  Carrier;op = op}

record RelationalMagma : Set₁ where
    field Carrier       : Set
    field op        : Carrier → Carrier → Carrier
    toType      : let View X = X in View Type ; toType = record {Carrier = Carrier}
    field _≈_       : Carrier → Carrier → Set
    toMagma     : let View X = X in View Magma ;    toMagma = record {Carrier =
    ↪  Carrier;op = op}
    field cong      : ∀ x x′ y y′ → _≈_ x x′ → _≈_ y y′ → _≈_ (op x x′) (op y y′)
    toRawRelationalMagma        : let View X = X in View RawRelationalMagma ;
    ↪  toRawRelationalMagma = record {Carrier = Carrier;op = op;_≈_ = _≈_}
```

( The `let View X = X in View` $\boxed{\cdots}$ clauses are a part of the user implementation of `extended-by`; they are used as markers to indicate that a declaration is a *view* and so should not be an element of the current view constructed by a call to `extended-by`. )

Hence, we have a formal approach to the idea that **each piece of mathematical knowledge should be formalised only once** [DBLP:conf/aisc/GrabowskiS10].

In conjunction with `postulating`, the `extended-by` variational makes it **tremendously easy to build fine-grained hierarchies** since at any stage in the hierarchy we have views to parent stages (unless requested otherwise) *and* the hierarchy structure is *hidden* from end-users. That is to say, ignoring the views, the above initial declaration of $\texttt{CommutativeMonoid}_0$ is identical to the `CommutativeMonoid` package obtained by using variationals, as follows.

Building fine-grained hierarchies with ease

```
PackageFormer Empty : Set1 where {- No elements -}
Type                 = Empty               extended-by "Carrier : Set"
Magma                = Type                extended-by "_·_ : Carrier → Carrier →
↪  Carrier"
Semigroup            = Magma               postulating "_·_" "associative"
LeftUnitalSemigroup  = Semigroup           postulating "_·_" "left-unit"  :using "𝕀"
Monoid               = LeftUnitalSemigroup postulating "_·_" "right-unit" :using "𝕀"
CommutativeMonoid    = Monoid              postulating "_·_" "commutative"
```

Of-course, one can continue to build packages in a monolithic fashion, as shown below.

```
GroupR = MonoidR extended-by "_⁻¹ : Carrier → Carrier; left⁻¹ : ∀ {x} → (x ⁻¹) ·
↪  x ≡ 𝕀; right⁻¹ : ∀ {x} → x · (x ⁻¹) ≡ 𝕀" ⟴ record
```

## 4.3.3 Renaming

From an end-user perspective, our CommutativeMonoid has one flaw: Such monoids are frequently written *additively* rather than multiplicatively. Such a change can be rendered conveniently:

Renaming Example

```
{-700
AbealianMonoidR = CommutativeMonoidR renaming "_·_ to _+_"
-}
```

An Abealian monoid is *both* a commutative monoid and also, simply, a monoid. The above declaration freely maintains these relationships: The resulting record comes with a new projection toCommutativeMonoidR, and still has the *inherited* projection toMonoidR.

There are a few reasonable properties that a renaming construction should support. Let us briefly look at the properties of renaming.

### Relationship to Parent Packages

Dual to extended-by which can construct (retract) views **to parent** modules mechanically, renaming constructs (coretract) views **from parent** packages. That is, it has an optional argument :adjoin-coretract which can be provided with t to use a default name or provided with a string to use a desired name for the inverse part of a projection, fromMagma below.

Adjoining coretracts —views from parent packages

```
{-700
Sequential = Magma renaming "op to _⨾_" :adjoin-coretract t
-}
```

Elaboration

```
record Sequential : Set₁ where
    field Carrier : Set
    field _⨾_     : Carrier → Carrier → Carrier

    toType : let View X = X in View Type
    toType = record {Carrier = Carrier}

    toMagma : let View X = X in View Magma
    toMagma = record {Carrier = Carrier;op = _⨾_}

    fromMagma : let View X = X in Magma → View Sequential
    fromMagma = λ g227742 → record {Carrier = Magma.Carrier g227742;_⨾_ = Magma.op
    ↪  g227742}
```

As the elaboration show, the user implementation of `renaming` makes use of *gensym*'s — generated symbolic names, "fresh variable names"— for λ-arguments to avoid name clashes.

## Commutativity

Since `renaming` and `extended-by` (including `postulating`) both adjoin retract morphisms, by default, we are lead to wonder about the result of performing these operations in sequence 'on the fly', rather than naming each application. Since `P renaming X ⟴ postulating Y` comes with a retract `toP` via the `renaming` and another, distinctly defined, `toP` via `postulating`, we have that the operations commute if *only* the first permits the creation of a retract. Below is a concrete example wherein we may replace

```
renaming "_·_ to _⊔_" ⟴ postulating "_⊔_" "idempotent"
```

with

```
postulating "_⊔_" "idempotent" ⟴ renaming "_·_ to _⊔_"
```

and still end up with the same elaboration, up to order of constituents.

```
{-700
IdempotentMagma  = Magma renaming "_·_ to _⊔_" ⟴ postulating "_⊔_" "idempotent"
↪  :adjoin-retract nil ⟴ record
-}
```

It is important to realise that the renaming and postulating combinators are *user-defined*, and could have been defined without adjoining a retract by default; consequently, we would have **unconditional commutativity of these combinators**. The user can make these alternative combinators as follows:

Alternative 'renaming' and 'postulating' —with an example use

```
{-700
𝒱-renaming′ by = renaming 'by :adjoin-retract nil
𝒱-postulating′ p bop (using) = postulating 'p 'bop :using 'using :adjoin-retract
↪  nil

IdempotentMagma′ = Magma postulating′ "_⊔_" "idempotent" ⟴ renaming′ "_·_ to _⊔_"
↪  ⟴ record
-}
```

### Simultaneous Textual Substitution

As expected, simultaneous renaming works too.

```
{-700
PackageFormer Two : Set₁ where
  Carrier : Set
  𝟘       : Carrier
  𝟙       : Carrier

TwoR = Two record ⟴ renaming′ "𝟘 to 𝟙; 𝟙 to 𝟘"
-}
```

`TwoR` is just `Two` but as an Agda `record`, so it typechecks.

### Involution; self-inverse

Finally, renaming is an invertible operation —ignoring the adjoined retracts, `Magma`$^{rr}$ is identical to `Magma`.

```
{-700
Magmaʳ  = Magma  renaming "_·_  to op"
Magmaʳʳ = Magmaʳ renaming "op   to _·_"
-}
```

### Do-it-yourself

Finally, to demonstrate the accessibility of the system, we show how a generic renaming operation can be defined swiftly using the extended set of meta-primitives mentioned in the lower part of Table ??. Instead of `renaming` elements *one at a time*, suppose we want to be able to uniformly `rename` all elements in a package. That is, given a function `f` on strings, we want to map over the name component of each element in the package. This is easily done with the following declaration.

Tersely forming a new variational

```
𝒱-rename f = map (λ element → (map-name (λ nom → (funcall f nom))) element)
```

Perhaps the main point of the above definition that may be unexpected to the Agda programmer is that Lisp function calls are of the form (`function` $\texttt{arg}_0$ $\texttt{arg}_1$ ... $\texttt{arg}_n$).

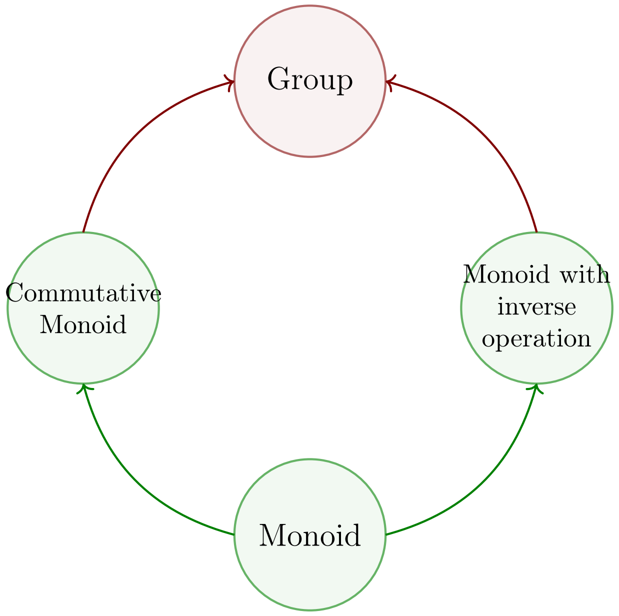
## 4.3.4 Unions/Pushouts (and intersections)

But even with these features, using `GroupR` from above, we would find ourselves writing:

```
{-700
CommutativeGroupR₀  = GroupR extended-by "comm : {x y : Carrier} →  x · y  ≡  y ·
↪  x" ⟴ record
-}
```

This is **problematic**: We lose the *relationship* that every commutative group is a commutative monoid. This is not an issue of erroneous hierarchical design: From `Monoid`, we could orthogonally add a commutativity property or inverse operation; $\texttt{CommutativeGroupR}_0$ then closes this diamond-loop by adding both features, as shown in Figure ??. The simplest way to share structure is to union two presentations:

Figure 4.4: Given green, require red



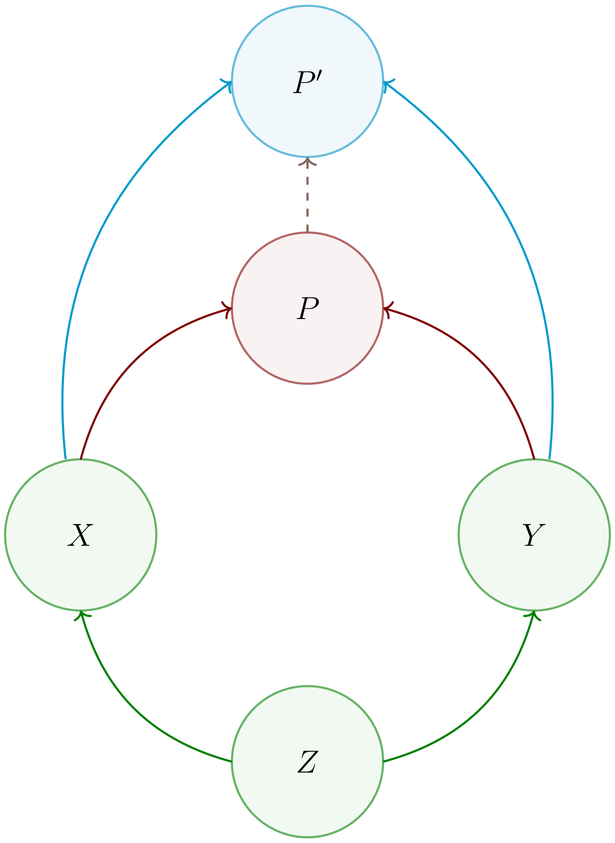
Unions of packages

```
{-700
CommutativeGroupR = GroupR union CommutativeMonoidR ⟴ record
-}
```

The resulting record, `CommutativeMonoidR`, comes with three derived fields —`toMonoidR, toGroupR, toCommutativeMonoidR`— that retain the results relationships with its hierarchical construction. This approach “works” to build a sizeable library, say of the order of 500 concepts, in a fairly economical way **tpc**. The union operation is an instance of a *pushout* operation, which consists of 5 arguments —three objects and two morphisms— which may be included into the `union` operation as optional keyword arguments. The more general notion of pushout is required if we were to combine `GroupR` with `AbealianMonoidR`, which have non-identical syntactic copies of `MonoidR`.

The pushout of $f : Z \to X$ and $g : Z \to Y$ is, essentially, the disjoint sum of $X$ and $Y$ where embedded elements are considered ‘indistinguishable’ when they share the same origin in $Z$ via the paths $f$ and $g$ —the pushout generalises the notion of *least upper bound* as shown in Figure **??** by treating each ‘$\to$’ as a ‘$\leq$’. Unfortunately, the resulting ‘indistinguishable’ elements $f(z) \approx g(z)$ are **actually distinguishable**: They may be the $f$-name or the $g$-name and a choice must be made as to which name is preferred since users actually want to refer to them later on. Hence, to be useful for library construction, the pushout construction actually requires at least another input function that provides canonical names to the supposedly ‘indistinguishable’ elements.

Figure 4.5: Given green, require red, such that every candidate cyan has a unique umber



Since a `PackageFormer` is essentially just a *signature* —a collection of typed names—, we can make a 'partial choice of pushout' to reduce the number of arguments from 6 to 4 by letting the typed-names object $Z$ be 'inferred' and encoding the canonical names function into the operations $f$ and $g$. The inputs functions $f, g$ are necessarily *signature morphisms* —mappings of names that preserve types— and so are simply lists associating names of $Z$ to names of $X$ and $Y$. If we instead consider $f' : Z' \leftarrow X$ and $g' : Z' \leftarrow Y$, in the *opposite direction*, then we may reconstruct a pushout by setting $Z$ to be common image of $f', g'$, and set $f, g$ to be inclusions. In-particular, the full identity of $Z'$ is not necessarily relevant for the pushout reconstruction and so it may be omitted. Moreover, the issue of canonical names is resolved: *If $x \in X$ is intended to be identified with $y \in Y$ such that the resulting element has $z$ as the chosen canonical name, then we simply require $f'\, x = z = g'\, y$.* An example is shown below in Figure **??**.

At first, a pushout construction needs 5 inputs, to be practical it further needs a function for canonical names for a total of 6 inputs. However, a pushout of $f : Z \to X$ and $g : Z \to Y$ is intended to be the 'smallest object $P$ that contains a copy of $X$ and of $Y$ sharing the common substructure $X$', and as such it outputs two functions $inj_1 : X \to P$, $inj_2 : Y \to P$ that inject the names of $X$ and $Y$ into $P$. If we realise $P$ as a record —a type of models— then the embedding functions are *reversed*, to obtain projections $P \to X$ and $P \to Y$: If we have a model of $P$, then we can forget some structure and rename via $f$ and $g$ to obtain models of $X$ and $Y$. For the resulting construction to be useful, these names could be automated such as $toX : P \to X$ and $toY : P \to Y$ but such a naming scheme does not scale —but we shall use it for default names. As such, we need two more inputs to the pushout construction so the names of the resulting output functions can be used later on. *Hence, a practical choice of pushout needs 8 inputs!*

Using the above issue to reverse the directions of $f, g$ via $f', g'$, we can infer the shared structure $Z$ and the canonical name function. Likewise, by using $toChild : P \to Child$ default-naming scheme, we may omit the names of the retract functions. If we wish to rename these retracts or simply omit them altogether, we make the *optional* arguments: Provide `:adjoin-retract`$_i$ `"new-function-name"` to use a new name, or `nil` instead of a string to omit the retract —as was done for `extended-by` earlier.

(Abridged) Pushout combinator with 6 optional arguments

```
(𝒱 union pf (renaming₁ "") (renaming₂ "") (adjoin-retract₁ t) (adjoin-retract₂ t)

 = "Union the elements of the parent PackageFormer with those of
    the provided PF symbolic name, then adorn the result with two views:
    One to the parent and one to the provided PF.

    If an identifer is shared but has different types, then crash.

    ADJOIN-RETRACTᵢ, for i : 1..2, are the optional names of the resulting
    views. Provide NIL if you do not want the morphisms adjoined.
    "
   :alter-elements (λ es →
     (let* ((p (symbol-name 'pf))
            (es₁ (alter-elements es renaming renaming₁ :adjoin-retract nil))
            (es₂ (alter-elements ($elements-of p) renaming renaming₂
                                 :adjoin-retract nil))
            (es′ (-concat es₁ es₂))
            (name-clashes (loop for n in (find-duplicates (mapcar #'element-name
            ↪  es′))
                                for e = (--filter (equal n (element-name it)) es′)
                                unless (--all-p (equal (car e) it) e)
                                collect e))
            (er₁ (if (equal t adjoin-retract₁) (format "to%s" $parent)
                   adjoin-retract₁))
            (er₂ (if (equal t adjoin-retract₂) (format "to%s" p)
                   adjoin-retract₂)))

       ;; Error on name clashes; unabridged version has a mechanism to “fix
       ↪  conflicts”
       ;; The unabridged version accounts for name clashes on retracts as well.
       (if name-clashes
            (-let [debug-on-error nil]
              (error "%s = %s union %s \n\n\t\t → Error:
                      Elements ‘‘%s’’ conflict!\n\n\t\t\t%s"
                      $name $parent p (element-name (caar name-clashes))
                      (s-join "\n\t\t\t" (mapcar #'show-element (car
                      ↪  name-clashes))))))

   ;; return value
   (-concat es′
            (and adjoin-retract₁ (not er₁) (list (element-retract $parent es :new
            ↪  es₁ :name adjoin-retract₁)))
            (and adjoin-retract₂ (not er₂) (list (element-retract p ($elements-of p)
            ↪  :new es₂ :name adjoin-retract₂)))))))
```

The reader is not meant to understand the (abridged[5]) definition provided here, however we

[5]The unabridged definition, on the PackageFormer webpage, has more features. In particular, it accepts additional keyword toggles that dictate how it should behave when name clashes occur; e.g., whether it should halt and report the name clash or whether it should silently perform a name change, according to another provided argument. The additional flexibility is useful for rapid experimentation.

present a few implementation remarks and wish to emphasise that this definition is **not built in**, and so the user could have, for example, provided a faster implementation by omitting checks for name clashes.

1. Since the systems allows optional keyword arguments, the first line declares only a context name, `pf`, is mandatory and the remaining arguments to a pushout are 'inferred' unless provided.
2. The second line documents this new user-defined variational; the documentation string is attached as a tooltip to all instances of the phrase `union`.
3. Given `f, g` as `renaming`$_i$, we apply the renaming variational on the elements of the implicit context (to this variational) and to the given context `pf` to obtain two new element lists `e`$_i$.
4. We then adjoin retract elements `er`$_i$.
5. Finally, we check for name clashes and handle them appropriately.

The user manual contains full details and an implementation of intersection, pullback, as well. We now turn to some examples of this construction to see it in action; in particular, here is an example which mentions all arguments, optional and otherwise. Besides the specification's elaboration, we also provide a **commutative** diagram, Figure ??, that *informally* carries out the `union` construction.

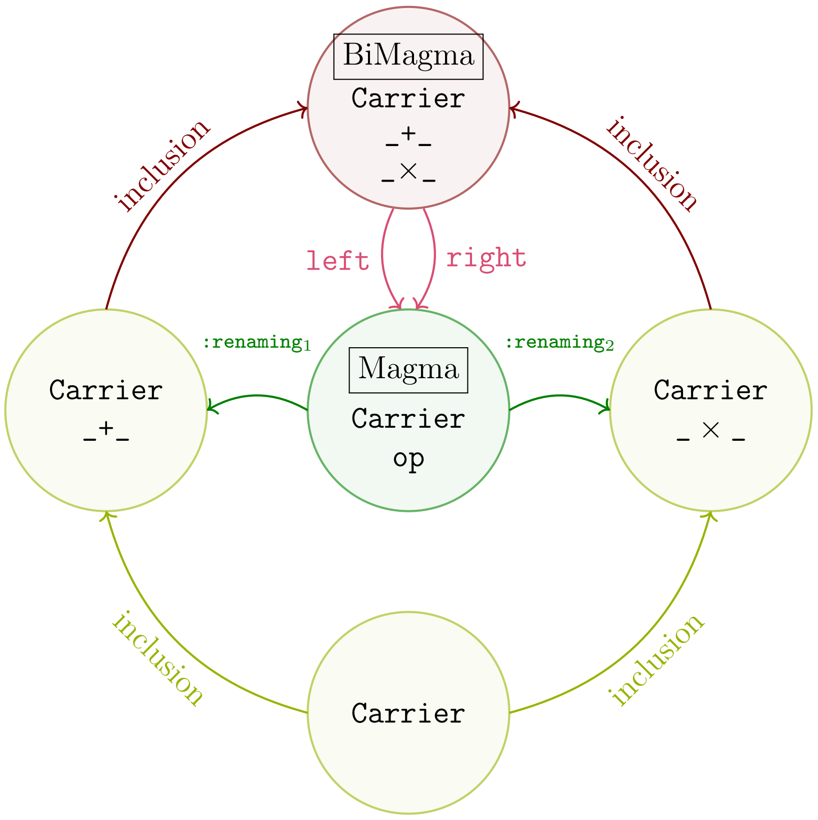
Bimagmas: Two magmas sharing the same carrier

```
{-700
TwoBinaryOps = Magma union Magma :renaming₁ "op to _+_" :renaming₂ "op to _×_"
↪  :adjoin-retract₁ "left" :adjoin-retract₂ "right"
-}
```

Elaboration

```
record TwoBinaryOps : Set₁ where
    field Carrier : Set
    field _+_     : Carrier → Carrier → Carrier

    toType : let View X = X in View Type
    toType = record {Carrier = Carrier}

    field _×_     : Carrier → Carrier → Carrier

    left : let View X = X in View Magma
    left = record {Carrier = Carrier;op = _+_}

    right : let View X = X in View Magma
    right = record {Carrier = Carrier;op = _×_}
```

Figure 4.6: Given green, yield yellow, require red, form fuchsia



Remember, *this particular user implementation* realises

$$\texttt{X}_1 \ \texttt{union} \ \texttt{X}_2 \ \texttt{:renaming}_1 \ \texttt{f}\boxed{'} \ \texttt{:renaming}_2 \ \texttt{g}\boxed{'}$$

as the pushout of the inclusions $\texttt{f}\boxed{'} \ \texttt{X}_1 \ \boxed{\cap} \ \texttt{g}\boxed{'} \ \texttt{X}_2 \ \boxed{\hookrightarrow} \ \texttt{X}_i$ where the source is the set-wise intersection of *names*. Moreover, when either $\texttt{renaming}_i$ is omitted, it defaults to the identity function.

We now turn to useful properties of the user-defined `union` variational.

### Idempotence —Set Union

The next example is one of the reasons the construction is named 'union' instead of 'pushout': It's idempotent, if we ignore the addition of the retract.

```
{-700
MagmaAgain   = Magma union Magma
-}
```

```
record MagmaAgain : Set₁ where
    field Carrier : Set
    field op      : Carrier → Carrier → Carrier

    toType : let View X = X in View Type
    toType = record {Carrier = Carrier}

    toMagma : let View X = X in View Magma
    toMagma = record {Carrier = Carrier;op = op}
```

Of note is that this is essentially the previous bi-magma example —Figure **??**— *but* we are not distinguishing —via $\texttt{:renaming}_i$— the two instances of `Magma`.

### Disjointness —Categorial Sums

We may perform disjoint sums —simply distinguish all the names of one of the input objects.

```
{-700
Magma′     = Magma primed ⟴ record
SumMagmas = Magma union Magma′ :adjoin-retract₁ nil ⟴ record
-}
```

Elaboration

```
record SumMagmas : Set₁ where
    field Carrier  : Set
    field op       : Carrier → Carrier → Carrier

    toType         : let View X = X in View Type
    toType = record {Carrier = Carrier}

    field Carrier′ : Set
    field op′      : Carrier′ → Carrier′ → Carrier′

    toType′ : let View X = X in View Type
    toType′ = record {Carrier = Carrier′}

    toMagma : let View X = X in View Magma
    toMagma = record {Carrier = Carrier′;op = op′}

    toMagma′ : let View X = X in View Magma′
    toMagma′ = record {Carrier′ = Carrier′;op′ = op′}
```

Of note is that this is essentially the previous bi-magma example —Figure **??**— *but* we are not distinguishing the two instances of `Magma` 'on the fly' via `:renaming`$_i$ but instead making them disjoint beforehand using the following *informal* equation:

```
p primed  ≈  p :renaming (λ name ↦ name ++ "′")
```

## Support for Diamond Hierarchies

A common scenario is extending a structure, say `Magma`, into orthogonal directions, such as by making it operation associative or idempotent, then closing the resulting diamond by combining them, to obtain a semilattice. However, the orthogonal extensions may involve different names and so the resulting semilattice presentation can only be formed via pushout; below are three ways to form it.

Three ways to get to SemiLattice

```
{-700
Semigroup          = Magma postulating "_·_" "associative"
IdempotentMagma    = Magma renaming "_·_ to _⊔_" ⟴ postulating "_⊔_" "idempotent"
↪  :adjoin-retract nil

⊔-SemiLattice     = Semigroup union IdempotentMagma :renaming₁ "_·_ to _⊔_"
·-SemiLattice     = Semigroup union IdempotentMagma :renaming₂ "_⊔_ to _·_"
↑-SemiLattice     = Semigroup union IdempotentMagma :renaming₁ "_·_ to _↑_"
↪  :renaming₂ "_⊔_ to _↑_"
-}
```

**Application: Granular (Modular) Hierarchy for Rings**

We will close with the classic example of forming a ring structure by combining two monoidal structures. This example also serves to further showcase how using `postulating` can make for more granular, modular, developments.

```
{-700
Additive           = Magma renaming "_·_ to _+_" ⟴ postulating "_+_" "commutative"
↪  :adjoin-retract nil ⟴ record
Multiplicative     = Magma renaming "_·_ to _×_" :adjoin-retract nil ⟴ record
AddMult            = Additive union Multiplicative ⟴ record
AlmostNearSemiRing = AddMult ⟴ postulating "_×_" "distributiveˡ" :using "_+_" ⟴
↪  record
-}
```

Elaboration

```
record AlmostNearSemiRing : Set₁ where
    field Carrier : Set
    field _+_     : Carrier → Carrier → Carrier

    toType : let View X = X in View Type
    toType = record {Carrier = Carrier}

    toMagma : let View X = X in View Magma
    toMagma = record {Carrier = Carrier;op = _+_}

    field comm       : ∀ x y   → _+_ x y ≡ _+_ y x
    field _×_         : Carrier → Carrier → Carrier

    toAdditive : let View X = X in View Additive
    toAdditive = record {Carrier = Carrier;_+_ = _+_;comm = comm}

    toMultiplicative : let View X = X in View Multiplicative
    toMultiplicative = record {Carrier = Carrier;_×_ = _×_}

    field distˡ      : ∀ x y z → _×_ x (_+_ y z) ≡ _+_ (_×_ x y) (_×_ x z)
```

## 4.3.5 Duality

Maps between grouping mechanisms are sometimes called *views*, which are essentially an internalisation of the *variationals* in our system. A useful view is that of capturing the heuristic of *dual concepts*, e.g., by changing the order of arguments in an operation. That is, the dual, or opposite, of a binary operation `_·_ : X → Y → Z` is the operation `_·ᵒᵖ_ : Y → X → Z` defined by `x ·ᵒᵖ y = y · x`. Classically in Agda, duality is *utilised* as follows:

1. Define a *parameterised* module `R _·_` for the desired ideas **on** the operation `_·_`.
   - ◇ Concretely, say it defines the predicate `·-isLeftId e = (∀ x → e · x ≡ x)`.
2. Define a shallow (parameterised) module `Rᵒᵖ _·_` that essentially only opens `R _·ᵒᵖ_` and renames the concepts in `R` with dual names.
   - ◇ Continuing the concrete example, `Rᵒᵖ _·_` would essentially be

```
public open R _·_ renaming (·-isLeftId to ·-isRightId)
```

The Ubiquity of Duality

The RATH-Agda [**RATH**] library performs essentially this approach, for example for obtaining `UpperBounds` from `LowerBounds` in the context of an ordered set. Moreover, since Category Theory can serve as a foundational system of reasoning (logic) and implementation (programming), the idea of duality immediately applies to produce "two for one" theorems and programs.

Unfortunately, this means that any record definitions in `R` must have their field names be sufficiently generic to play *both* roles of the original and the dual concept. Admittedly, RATH-Agda's names are well-chosen; e.g., `value`, `bound`$_i$, `universal` to denote a `value` that is a lower/upper `bound` of two given elements, satisfying a least upper bound or greatest lower bound `universal` property. However, well-chosen names come at an upfront cost: One must take care to provide sufficiently generic names and account for duality at the outset, irrespective of whether one *currently* cares about the dual or not; otherwise when the dual is later formalised, then the names of the original concept must be refactored throughout a library and its users. This is not the case using `PackageFormer`. Consider the following heterogeneous algebra —which is essentially the main example of section 4.2 but missing the associativity field.

Left unital actions

```
{-700
PackageFormer LeftUnitalAction : Set₁ where
  Scalar : Set
  Vector : Set
  _·_      : Scalar → Vector → Vector
  𝟙        : Scalar
  leftId   : {x : Vector} → 𝟙 · x ≡ x

-- Let's reify this as a valid Agda record declaration
LeftUnitalActionR  = LeftUnitalAction ⟴ record
-}
```

Informally, one now 'defines' a right unital action by duality, flipping the binary operation and renaming `leftId` to be `rightId`. Such informal parlance is in-fact nearly formally, as the following:

Right unital actions —mechanically by duality

```
{-700
RightUnitalActionR = LeftUnitalActionR flipping "_·_" :renaming "leftId to rightId"
↪  ⟴ record
-}
```

Of-course the resulting representation is semantically identical to the previous one, and so it is furnished with a *toParent* mapping:

```
forget : RightUnitalActionR → LeftUnitalActionR
forget = RightUnitalActionR.toLeftUnitalActionR
```

Likewise, for the RATH-Agda library's example from above, to define semi-lattice structures by duality:

```
import Data.Product as P

{-700
PackageFormer JoinSemiLattice : Set₁ where
  Carrier : Set
  _⊑_      : Carrier → Carrier → Set

  refl    : ∀ {x}      → x ⊑ x
  trans   : ∀ {x y z} → x ⊑ y → y ⊑ z → x ⊑ z
  antisym : ∀ {x y}    → x ⊑ y → y ⊑ x → x ≡ y

  _⊔_     : Carrier → Carrier → Carrier
  ⊔-lub   : ∀ {x y z} → x ⊑ z → y ⊑ z → (x ⊔ y) ⊑ z
  ⊔-lub˘  : ∀ {x y z} → (x ⊔ y) ⊑ z  →  x ⊑ z  P.×  y ⊑ z

JoinSemiLatticeR = JoinSemiLattice record
MeetSemiLatticeR = JoinSemiLatticeR flipping "_⊑_" :renaming "_⊔_ to _⊓_; ⊔-lub to
↪  ⊓-glb"
-}
```

In this example, besides the map from meet semi-lattices to join semi-lattices, the types of the dualised names, such as `⊓-glb`, are what one would expect were the definition written out explicitly:

Checking the types of the duals

```
module woah (M : MeetSemiLatticeR) where
  open MeetSemiLatticeR M

  lub_dual_type : ∀ {x y z} → z ⊑ x → z ⊑ y → z ⊑ (x ⊓ y)
  lub_dual_type = ⊓-glb

  trans_dual_type : let _⊒_ = λ x y → y ⊑ x
                    in ∀ {x y z} → x ⊒ y → y ⊒ z → x ⊒ z
  trans_dual_type = trans
```

### 4.3.6 Extracting Little Theories

The `extended-by` variational allows Agda users to easily employ the *tiny theories* **little_theories**; **mathscheme** approach to library design: New structures are built from old ones by augmenting one concept at a time —as shown below— then one uses mixins such as `union` to obtain a complex structure. This approach lets us write a program, or proof, in a context that only provides what is *necessary* for that program-proof and nothing more. In this way, we obtain *maximal generality* for re-use! This approach can be construed as **The Interface Segregation Principle** [**old-design-patterns-solid**; **design_patterns_head_first**]: *No client should be forced to depend on methods it does not use.*

Tiny Theories Example

```
{-700
PackageFormer Empty : Set₁ where {- No elements -}
Type  = Empty extended-by "Carrier : Set"
Magma = Type  extended-by "_·_ : Carrier → Carrier → Carrier"
CommutativeMagma = Magma extended-by "comm : {x y : Carrier} →  x · y  ≡  y · x"
-}
```

However, life is messy and sometimes one may hurriedly create a structure, then later realise that they are being forced to depend on unused methods. Rather than throw a `not implemented` exception or leave them undefined, we may use the `keeping` variational to **extract the smallest well-formed sub-PackageFormer that mentions a given list of identifiers**. For example, suppose we quickly formed `Monoid` **monolithicaly** as presented at the start of section 4.3.1, but later wished to utilise other substrata. This is easily achieved with the following declarations.

Extracting Substrata from a Monolithic Construction

```
{-700
Empty′         = Monoid keeping ""
Type′          = Monoid keeping "Carrier"
Magma′         = Monoid keeping "_·_"
Semigroup′     = Monoid keeping "assoc"
PointedMagma′ = Monoid keeping "▯; _·_"
                 -- This is just keeping: Carrier; _·_; ▯
-}
```

Even better, we may go about deriving results —such as theorems or algorithms— in familiar settings, such as `Monoid`, only to realise that they are written in **settings more expressive than necessary**. Such an observation no longer need to be found by inspection, instead it may be derived mechanically.

Specialising a result from an expressive setting to the **minimal** necessary setting

```
{-700
LeftUnitalMagma = Monoid keeping "▯-unique" ⟴ record
-}
```

This expands to the following theory, minimal enough to derive `▯-unique`.

Elaboration

```
record LeftUnitalMagma : Set₁ where

  field
    Carrier : Set
    _·_     : Carrier → Carrier → Carrier
    ▯       : Carrier
    leftId  : {x : Carrier} → ▯ · x  ≡ x

  ▯-unique     : ∀ {e} (lid : ∀ {x} → e · x ≡ x) (rid : ∀ {x} → x · e ≡ x) → e
  ↪  ≡ ▯
  ▯-unique lid rid = ≡.trans (≡.sym leftId) rid
```

Surprisingly, in some sense, `keeping` let’s us apply the interface segregation principle, or ‘little theories’, **after the fact** —this is also known as *reverse mathematics*.

### 4.3.7 200+ theories —one line for each

*People should enter terse, readable, specifications that expand into useful, type-checkable, code that may be dauntingly larger in textual size.*

In order to demonstrate the **immediate practicality** of the ideas embodied by `PackageFormer`, we have implemented a list of mathematical concepts from universal algebra —which is useful to computer science in the setting of specifications. The list of structures is adapted from the source of a MathScheme library **tpc**; **mathscheme**, which in turn was inspired by web lists of Peter Jipsen, John Halleck, and many others from Wikipedia and nlab. Totalling over 200 theories which elaborate into nearly 1500 lines of typechecked Agda, this demonstrates that our systems works; the **750% efficiency savings** speak for themselves.

*The 200+ one line specifications and their ~1500 lines of elaborated typechecked Agda can be found on `PackageFormer`'s webpage.*

`https://alhassy.github.io/next-700-module-systems`

If anything, this elaboration demonstrates our tool as a useful engineering result. The main novelty being the ability for library users to extend the collection of operations on packages, modules, and then have it immediately applicable to Agda, an **executable** programming language.

Since the resulting **expanded code is typechecked** by Agda, we encountered a number of places where non-trivial assumptions accidentally got-by the MathScheme team. For example, in a number of places, an arbitrary binary operation occurred multiple times leading to ambiguous terms, since no associativity was declared. Even if there was an implicit associativity criterion, one would then expect multiple copies of such structures, one axiomatisation for each parenthesisation. Moreover, there were also certain semantic concerns about the design hierarchy that we think are out-of-place, but we chose to leave them as is —e.g., one would think that a "partially ordered magma" would consist of a set, an order relation, and a binary operation that is monotonic in both arguments; however, `PartiallyOrderedMagma` instead comes with a single monotonicity axiom which is only equivalent to the two monotonicity claims in the setting of a monoidal operation. Nonetheless, we are grateful for the source file provided by the MathScheme team.

Extensiblity

Unlike other systems, `PackageFormer` does not come with a static set of module operators —it grows dynamically, possibly by you, the user.

## 4.4 Contributions: From Theory to Practice

The `PackageFormer` implements the ideas of Chapters 2 and 3. As such, as an editor extension, it is mostly **language agnostic** and could be altered to work with other languages such as Coq [**coq_implementing_modules**], Idris [**idris_tdd**], and even Haskell [**DBLP:conf/haskell/LindleyM13**]. The `PackageFormer` implementation has the following useful properties.

1. Expressive & extendable specification language for the library developer.
    - ⋄ Our meta-primitives give way to the ubiquitous module combinators of Table **??**.
    - ⋄ E.g., from a theory we can derive its homomorphism type, signature, its termtype, etc; we generate useful constructions inspired from universal algebra and seen in the wild —see Chapter 3.
    - ⋄ An example of the freedom allotted by the extensible nature of the system is that combinators defined by library developers can, say, utilise auto-generated names when names are irrelevant, use 'clever' default names, and allow end-users to supply desirable names on demand using Lisps' keyword argument feature —see section 4.3.4.
2. Unobtrusive and a tremendously simple interface to the end user.
    - ⋄ Once a library is developed using (the current implementation of) `PackageFormer`, the end user only needs to reference the resulting generated Agda, without any knowledge of the existence of `PackageFormer`.
        - ◦ Generated modules are necessarily 'flattened' for typechecking with Agda — see section 4.3.1.
    - ⋄ We demonstrates how end-users can build upon a library by using *one line* specifications, by reducing over 1500 lines of Agda code to nearly 200 specifications using `PackageFormer` syntax.
3. Efficient: Our current implementation processes over 200 specifications in ~3 seconds; yielding typechecked Agda code *which* is what consumes the majority of the time.
4. Pragmatic: Common combinators can be defined for library developers, and be furnished with concrete syntax for use by end-users.
5. Minimal: The system is essentially invariant over the underlying type system; with the exception of the meta-primitive `:waist` which requires a dependent type theory to express 'unbundling' component fields as parameters.
6. Demonstrated expressive power *and* use-cases.
    - ⋄ Common boiler-plate idioms in the standard Agda library, and other places, are provided with terse solutions using the `PackageFormer` system.
        - ◦ E.g., automatically generating homomorphism types and wholesale renaming fields using a single function —see section 4.3.3.
    - ⋄ Over 200 modules are formalised as one-line specifications.
7. Immediately useable to end-users *and* library developers.

- ◇ We have provided a large library to experiment with —thanks to the MathScheme group for providing an adaptable source file.
- ◇ In the online user manual, we show how to formulate module combinators using a simple and straightforward subset of Emacs Lisp —a terse introduction to Lisp is provided.

Recall that we alluded —in the introduction to section 4.3— that we have a categorical structure consisting of `PackageFormers` as objects and those variationals that are signature morphisms. While this can be a starting point for a semantics for `PackageFormer`, we will instead pursue a *mechanised semantics*. That is, we shall encode (part of) the syntax of `PackageFormer` as Agda functions, thereby giving it not only a semantics but rather a life in a familar setting and lifting it from the status of *editor extension* to *language library*.

# 5 The Context Library

The PackageFormer framework is a useful tool to experiment with uncommon ways to package things together, but is relies on shuffling (untyped) strings and lacks a solid semantical basis. Instead of adding semantics after-the-fact, with the lessons learned from developing PackageFormer, we go on in this section to produce `Context`, an *extensible do-it-yourself module system for Agda* ***within*** *Agda.*

We will show an automatic technique for unbundling data at will; thereby resulting in *bundling-independent representations* and in *delayed unbundling*. Our contributions are to show:

1. Languages with sufficiently powerful type systems and meta-programming can conflate record and term datatype declarations into one practical interface. In addition, the contents of these grouping mechanisms may be function symbols as well as propositional invariants —an example is shown at the end of Section 5.2. We identify the problem and the subtleties in shifting between representations in Section 5.1.

2. Parameterised records can be obtained on-demand from non-parameterised records (Section 5.2) .

   ◇ As with $\texttt{Magma}_0$, the traditional approach **coq_cat_experiences** to unbundling a record requires the use of transport along propositional equalities, with trivial `refl`exivity proofs —via the Σ-padding anti-pattern of Section 3.1.3. In Section 5.2, we develop a combinator, `_:waist_`, which removes the boilerplate necessary at the type specialisation location as well as at the instance declaration location.

3. Programming with fixed-points of unary type constructors can be made as simple as programming with term datatypes (Section 5.3).

4. Astonishingly, we mechanically regain ubiquitous data structures such as `ℕ`, `Maybe`, `List` as the term datatypes of simple pointed and monoidal theories (Section 5.4).

As an application, in Section 5.5 we show that the resulting setup applies as a semantics for declarative pre-processing `PackageFormer` tool —which also accomplishes the above tasks.

For brevity, and accessibility, the definitions in this chapter are presented in an informal form alongside a concrete implementation *without* explanation of implementation details.

A complicated Agda macro

```
accessible dashed pseudo-code
```

Code

```
    ... actual Agda implementation,
        requiring intimate familarity with reflection in Agda ...
```

The informal form is presented with the understanding that such functions need to be extended **homomorphically** over all possible term constructors of the host language. Enough is shown to communicate the techniques and ideas, as well as to make the resulting library usable. The details, which users do not need to bother with, are nonetheless presented so as to show how accessible these techniques are —in that, they do not require more than 15 lines per core concept.

Chapter Contents

## 5.1 The Problems

Let us begin anew by briefly reviewing the main problems, but this time directly using Agda as the language of discourse.

There are a number of problems when packaging up data, with the number of parameters being exposed being the pivotal concern. To exemplify the distinctions at the type level as more parameters are exposed, consider the following approaches to formalising a dynamical system —a collection of states, a designated start state, and a transition function.

Dynamical Systems

```
record DynamicSystem₀ : Set₁ where
  field
    State  : Set
    start  : State
    next   : State → State

record DynamicSystem₁ (State : Set) : Set where
  field
    start : State
    next  : State → State

record DynamicSystem₂ (State : Set) (start : State) : Set where
  field
    next : State → State
```

Each $\texttt{DynamicSystem}_i$ is a type constructor of `i`-many arguments; but it is **the types of these constructors that provide insight into the sort of data they contain** as shown in the following table and discussed in Sections 3.1.3 and 3.1.

| Type | Kind |
|---|---|
| $\texttt{DynamicSystem}_0$ | $\texttt{Set}_1$ |
| $\texttt{DynamicSystem}_1$ | `Π X : Set • Set` |
| $\texttt{DynamicSystem}_2$ | `Π X : Set • Π x : X • Set` |

Recall, say from Section 2.8.1, that we refer to the concern of moving from a record to a parameterised record as **the unbundling problem packaging_mathematical_structures**. For example, moving from the *type* $\texttt{Set}_1$ to the *function type* `Π X : Set • Set` gets us from $\texttt{DynamicSystem}_0$ to something resembling $\texttt{DynamicSystem}_1$, which we arrive at if we can obtain a *type constructor* `λ X : Set • ⋯`. We shall refer to the latter change as *reïfication* since the result is more concrete: It can be applied. This transformation will be denoted by $\Pi{\to}\lambda$. To clarify this subtlety, consider the following forms of the polymorphic identity function. Notice that $\texttt{id}_i$ *exposes* $i$-many details at the type level to indicate the sort of data it consists of. However, notice that $\texttt{id}_0$ is a **type of functions** whereas $\texttt{id}_1$ is a

**function on types**. Indeed, the latter two are derived from the first one: $\texttt{id}_{i+1}$ = Π→λ $\texttt{id}_i$. These equations are true by reflexivity, as shown below.

Polymorphic Identity Functions

```
id₀ : Set₁
id₀ = Π X : Set • Π e : X • X

id₁ : Π X : Set • Set
id₁ = λ (X : Set) → Π e : X • X

id₂ : Π X : Set • Π e : X • Set
id₂ = λ (X : Set) (e : X) → X

{- Surprisingly, the latter are derivable from the former -}

_ : id₁ ≡ Π→λ id₀
_ = refl

_ : id₂ ≡ Π→λ id₁
_ = refl
```

Of course, there is also the need for descriptions of values, which leads to term datatypes. We shall refer to the shift from record types to algebraic data types as **the termtype problem**. Our aim is to obtain all of these notions —of ways to group data together— from a single user-friendly context declaration, using monadic notation.

## 5.2 Monadic Notation

There is little use in an idea that is difficult to use in practice. As such, we conflate records and termtypes by starting with an ideal syntax they would share, then derive the necessary artefacts that permit it. As discussed at the start of the chapter, our choice of syntax is monadic do-notation [**DBLP:journals/iandc/Moggi91**; **DBLP:conf/haskell/MarlowJKM16**]:

Idealised syntax for one source of truth

```
DynamicSystem : Context ℓ₁
DynamicSystem = do State ← Set
                   start ← State
                   next  ← (State → State)
                   End
```

Here `Context`, `End`, and the underlying monadic bind operator are unknown. Since we want to be able to *expose* a number of fields at will, we may take `Context` to be types indexed by a number denoting exposure. Moreover, since records are product types, we expect there to be a recursive definition whose base case will be the identity of products, the unit type 𝟙 —which corresponds to ⊤ in the Agda standard library and to `()` in Haskell. The following table shows example exposure 'waists' for the `DynamicSystem` context.

Elaborations of `DynamicSystem` at various exposure levels

# +caption: Elaborations of DynamicSystem at various exposure levels

| Exposure | Elaboration |
|---|---|
| 0 | `Σ State : Set • Σ start : X • Σ next : State → State • 𝟙` |
| 1 | `Π State : Set • Σ start : X • Σ next : State → State • 𝟙` |
| 2 | `Π State : Set • Π start : X • Σ next : State → State • 𝟙` |
| 3 | `Π State : Set • Π start : X • Π next : State → State • 𝟙` |

With these elaborations of `DynamicSystem` to guide the way, we resolve two of our unknowns.

Context and End

```
{- "Contexts" are exposure-indexed types -}
Context = λ ℓ → ℕ → Set ℓ

{- Every type can be used as a context -}
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S

{- The "empty context" is the unit type -}
End : ∀ {ℓ} → Context ℓ
End = ‘ 𝟙
```

It remains to identify the definition of the underlying bind operation `>>=`. Usually, for a type constructor `m`, bind is typed `∀ {X Y : Set} → m X → (X → m Y) → m Y`. It allows one to "extract an `X`-value for later use" in the `m Y` context. Since our `m = Context` is from levels to types, we need to slightly alter bind's typing.

Defining Bind —First Attempt

```
_>>=_ : ∀ {a b}
      → (Γ : Context a)
      → (∀ {n} → Γ n → Context b)
      → Context (a ⊍ b)
(Γ >>= f) zero    = Σ γ : Γ 0 • f γ 0
(Γ >>= f) (suc n) = Π γ : Γ n • f γ n
```

The definition here accounts for the current exposure index: If zero, we have *record types*, otherwise *function types*. Using this definition, the above dynamical system context would need to be expressed using the lifting quote operation.

**The extensibility of `Context` is provided by the definition of bind: Rather than $\Sigma$ and $\Pi$, users may use or augment the framework in other forms —e.g., $\Pi^w$, $\mathcal{W}$, or let⋯in⋯ (as shown in $\mathcal{N}_1{}'$ below) *or combinations thereof.**

Example Use

```
‘ Set >>= λ State
      → ‘ State >>= λ start
             → ‘ (State → State) >>= λ next
                                → End

{- or -}

do State ← ‘ Set
   start ← ‘ State
   next  ← ‘ (State → State)
   End
```

Interestingly **Bird_2009**; **DBLP:conf/hopl/HudakHJW07**, use of `do`-notation in preference to bind, `>>=`, was suggested by John Launchbury in 1993 and was first implemented by Mark Jones in Gofer. Anyhow, with our goal of practicality in mind, we shall "build the lifting quote into the definition" of bind:

The Definition of Bind

```
_>>=_ : ∀ {a b}
      → (Γ : Set a)   -- Main difference
      → (Γ → Context b)
      → Context (a ⊎ b)
(Γ >>= f) zero    = Σ γ : Γ • f γ 0
(Γ >>= f) (suc n) = Π γ : Γ • f γ n
```

With this definition, the above declaration `DynamicSystem` typechecks. However, we do *not* have an isomorphism `DynamicSystem` $i \cong$ `DynamicSystem`$_i$, instead `DynamicSystem` $i$ are "factories": Given $i$-many arguments, a product value is formed. What if we want to *instantiate* some of the factory arguments ahead of time?

Factories and Instantiation —Natural numbers form a dynamic system

```
𝒩₀ : DynamicSystem 0   {- See the above elaborations  -}
𝒩₀ = ℕ , 0 , suc , tt

𝒩₁ : DynamicSystem 1
𝒩₁ = λ State → ??? {- Impossible to complete if “State” is empty! -}

{- “Instantiaing” State to be ℕ in “DynamicSystem 1” -}
𝒩₁′ : let State = ℕ in Σ start : State   • Σ s : (State → State)   • 𝟙
𝒩₁′ = 0 , suc , tt
```

To get from $\mathcal{N}_1$ to $\mathcal{N}_1'$, it seems what we need is a method, say Π→λ, that takes a Π-type and transforms it into a λ-expression. One could use a universe, an algebraic type of codes denoting types, to define Π→λ. However, one can no longer then easily use existing types since they are not formed from the universe's constructors, thereby resulting in duplication of existing types via the universe encoding. This is neither practical nor pragmatic. As such, we are left with pattern matching on the language's type formation primitives as the only reasonable approach. The method Π→λ is thus a macro[1] that acts on the syntactic term representations of types. Below is the main transformation.

Π→λ

```
Π→λ (Π a ∶ A • τ) = (λ a ∶ A • τ)
```

Source —for the interested reader

```
Π→λ-helper : Term → Term
Π→λ-helper (pi  a b)         = lam visible b
Π→λ-helper (lam a (abs x y)) = lam a (abs x (Π→λ-helper y))
{-# CATCHALL #-}
Π→λ-helper x = x

macro
  Π→λ : Term → Term → TC Unit.⊤
  Π→λ tm goal = normalise tm >>=ₘ λ tm′ → unify (Π→λ-helper tm′) goal
```

That is, we walk along the term tree replacing (consecutive) occurrences of Π with λ; as shown in the following example.

[1]A *macro* is a function that manipulates the abstract syntax trees of the host language. In particular, it may take an arbitrary term, shuffle its syntax to provide possibly meaningless terms or terms that could not be formed without pattern matching on the possible syntactic constructions. An up to date and gentle introduction to reflection in Agda can be found at [**gentle-intro-to-reflection**].

Example use of Π→λ

```
  Π→λ (Π→λ (DynamicSystem 2))
≡{- Definition of DynamicSystem at exposure level 2 -}
  Π→λ (Π→λ (Π X : Set • Π s : X  • Σ n : X → X  • 𝟙))
≡{- Definition of Π→λ -}
  Π→λ (λ X : Set • Π s : X  • Σ n : X → X  • 𝟙)
≡{- Homomorphy of Π→λ -}
  λ X : Set • Π→λ (Π s : X  • Σ n : X → X  • 𝟙)
≡{- Definition of Π→λ -}
  λ X : Set • λ s : X  • Σ n : X → X  • 𝟙
```

For practicality, we define a macro `_:waist_` acting on contexts that *repeats* Π→λ a number of times in order to lift a number of field components to the parameter level.

Waist

$$\tau \text{ :waist n } = \Pi\boxed{\to}\lambda^n\ (\tau \text{ n})$$
$$f^0 \text{ x } = \text{ x}$$
$$f^{n\boxed{+1}} \text{ x } = f^n \text{ (f x)}$$

Source —for the interested reader

```
waist-helper : ℕ → Term → Term
waist-helper zero t    = t
waist-helper (suc n) t = waist-helper n (Π→λ-helper t)

macro
  _:waist_ : Term → Term → Term → TC Unit.⊤
  _:waist_ t n goal =      normalise (t app n)
                       >>=ₘ λ t′ → unify (waist-helper (toℕ n) t′) goal
```

We can now “fix arguments ahead of time”. Before such demonstration, we need to be mindful of our practicality goals: One declares a grouping mechanism with `do` $\boxed{\ldots}$ `End`, which in turn has its instance values constructed with ⟨ … ⟩, as defined below.

Syntactic Sugar for Context Values

```
-- Expressions of the form "··· , tt" may now be written "⟨ ··· ⟩"
infixr 5 ⟨ _⟩
⟨⟩ : ∀ {ℓ} → 𝟙 {ℓ}
⟨⟩ = tt

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

_⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × (𝟙 {ℓ})
s ⟩ = s , tt
```

The following instances of grouping types demonstrate how information moves from the body level to the parameter level.

Unbundling: Lifting Fields into Parameters

```
𝒩⁰ : DynamicSystem :waist 0
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : (DynamicSystem :waist 1) ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : (DynamicSystem :waist 2) ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : (DynamicSystem :waist 3) ℕ 0 suc
𝒩³ = ⟨⟩
```

Using `:waist` $i$ we may fix the first $i$-parameters ahead of time. Indeed, the type `(DynamicSystem :waist 1) ℕ` is *the type of dynamic systems over carrier* ℕ, whereas `(DynamicSystem :waist 2) ℕ 0` is *the type of dynamic systems over carrier* ℕ *and start state 0.*

Examples of the need for such on-the-fly unbundling can be found in numerous places in the Haskell standard library. For instance, the standard libraries **data_monoid** have two isomorphic copies of the integers, called `Sum` and `Product`, whose reason for being is to distinguish two common monoids: The former is for *integers with addition* whereas the latter is for *integers with multiplication*. An orthogonal solution would be to use contexts:

Monoids without commitment

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              _⊕_     ← (Carrier → Carrier → Carrier)
              Id      ← Carrier
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

With this context, `(Monoid` $\ell_0$ `:waist 2) M _⊕_` is the type of monoids over *particular* types `M` and *particular* operations `_⊕_`. Of-course, this is orthogonal, since traditionally unification on the carrier type `M` is what makes typeclasses and canonical structures **coq_canonical_tutorial** useful for ad-hoc polymorphism.

## 5.3 Termtypes as Fixed-points

We have a practical monadic syntax for possibly parameterised record types that we would like to extend to termtypes. As discussed in the previous section, we could alter the bind operator to account for $\mathcal{W}$-types, but we shall present a different technique so as to avoid "making bind do too much". Algebraic data types are a means to declare concrete representations of the least fixed-point of a functor; see **DBLP:journals/jfp/Swierstra08** for more on this idea. In particular, the description language $\mathbb{D}$ for dynamical systems, below, declares concrete constructors for a fixpoint of a certain functor $\mathcal{D}$; i.e., $\mathbb{D} \cong$ `Fix` $\mathcal{D}$ where:

ADTs and Functors

```
data 𝔻 : Set where
    startD : 𝔻
    nextD  : 𝔻 → 𝔻

𝒟 : Set → Set
𝒟 = λ (D : Set) → 𝟙 ⊎ D

data Fix (F : Set → Set) : Set where
  μ : F (Fix F) → Fix F
```

The problem is whether we can derive $\mathcal{D}$ from `DynamicSystem`. Let us attempt a quick calculation sketching the necessary transformation steps (informally expressed via "⇝"):

From Contexts to Fixed-points: A Roadmap

```
  do S ← Set; s ← S; n ← (S → S); End
⇝{- Use existing interpretation to obtain a record. -}
  Σ S : Set • Σ s : S • Σ n : (S → S) • 𝟙
⇝{- Pull out the carrier, “:waist 1”,
    to obtain a type constructor using “Π→λ”. -}
  λ S : Set • Σ s : S • Σ n : (S → S) • 𝟙
⇝{- Termtype constructors target the declared type,
    so only their sources matter. E.g., ‘s : S’ is a
    nullary constructor targeting the carrier ‘S’.
    This introduces 𝟙 types, so any existing
    occurances are dropped via 𝟘. -}
  λ S : Set • Σ s : 𝟙 • Σ n : S • 𝟘
⇝{- Termtypes are sums of products. -}
  λ S : Set •       𝟙   ⊎      S  ⊎ 𝟘
⇝{- Termtypes are fixpoints of type constructors. -}
  Fix (λ S • 𝟙 ⊎ S)  -- i.e., 𝒟
```

Since we may view an algebraic data-type as a fixed-point of the functor obtained from the union of the sources of its constructors, it suffices to treat the fields of a record as constructors, then obtain their sources, then union them. That is, since algebraic-datatype constructors necessarily target the declared type, they are determined by their sources. For example, considered as a unary constructor `op : A → B` targets the termtype `B` and so its source is `A`. Hence, we can form the `termtype` of a context as the `Fix`-point of the sum —using Σ→⊎— of the `sources` of the context, as shown below. Where the operation Σ→⊎ rewrites dependent-sums into disjoint sums, which requires the second argument to lose its reference to the first argument which is accomplished by ↓↓; further details can be found in the appendices.

```
sources (λ x : (Π a : A • Ba) • τ) = (λ x : A • sources τ)
sources (λ x : A              • τ) = (λ x : 𝟙 • sources τ)

↓↓ τ = “reduce all de-bruijn indices within τ by 1”

Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (↓↓ Ba)

termtype τ = Fix (Σ→⊎ (sources τ))
```

Before moving to an instructive **use** of this combinator, let us touch a bit on the details of its **formation**.

### 5.3.1 The `termtype` combinator

Using the guiding calculation above, we shall work up to the desired functor $\mathcal{D}$ by *implementing* each stage $i$ of the calculation and showing the approximation $\mathtt{D}_i$ of the functor $\mathcal{D}$

at that stage.

### Stage 1: Records

The first step is already possible, using the existing `Context` setup.

Building up to the `termtype` combinator

```
D₁ = DynamicSystem 0

1-records : D₁ ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • ⊤)
1-records = refl
```

### Stage 2: Parameterised Records

The second step is also already implemented, using the existing `_:waist_` mechanism.

Building up to the `termtype` combinator

```
D₂ = DynamicSystem :waist 1

2-funcs : D₂ ≡ (λ (X : Set) → Σ z : X • Σ s : (X → X) • ⊤)
2-funcs = refl
```

### Stage 3: Sources

As per the informal description of `sources` in the guiding calculation, we reinforce the idea with a number of desired test cases —as usual, formal machine checked test cases and Agda code can be found on the thesis repository. In particular, we make a **design decision** for the resulting `termtype` combinator: Types starting with implicit arguments are *invariants*, not *constructors* —and so are dropped from the resulting ADT by replacing them with the empty type '𝟘'.

Example uses of `sources`

| $\tau$ | `sources` $\tau$ |
|---|---|
| `Src → Tgt` | `Src` |
| `Σ f : (Src → Tgt) • Bdy` | `Σ x : Src • Bdy` |
| $\tau_1$ `→ ⋯ →` $\tau_n$ | $\tau_1$ `× ⋯ ×` $\tau_{n-1}$ `× 𝟙` |
| `Σ f :` $\tau_1$ `→ ⋯ →` $\tau_n$ `• Bdy` | `Σ x : (`$\tau_1$ `× ⋯ ×` $\tau_{n-1}$`) • Bdy` |
| `∀ {x : ℕ} → x ≡ x` | `𝟘` |
| `(∀ {x y z : ℕ} → x ≡ y)` | `𝟘` |

The third stage can now be formed.

Building up to the `termtype` combinator

```
D₃ = sources D₂

3-sources : D₃ ≡ λ (X : Set) → Σ z : 𝟙 • Σ s : X • 𝟘
3-sources = refl
```

With the following definitions.

sources

```
sources (λ x : (Π a : A ● Ba) ● τ) = (λ x : A ● sources τ)
sources (λ x : A          ● τ) = (λ x : 𝟙 ● sources τ)
```

Building up to the **termtype** combinator

```
sources₀ : Term → Term
sources₀ (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) =
    def (quote _×_) (vArg A
                    :: vArg (def (quote _×_)
                                (vArg (var-dec Ba)
                                     :: vArg (var-dec (var-dec (sources₀ Cab)))
                                     ↪  :: []))
                    :: [])
sources₀ (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 𝟘
sources₀ (Π[ x : arg i A ] Bx) = A
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources₀ t = quoteTerm 𝟙

{-# TERMINATING #-}
sources₁ : Term → Term
sources₁ (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 𝟘
sources₁ (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) = def (quote _×_) (vArg A
↪  ::
  vArg (def (quote _×_) (vArg (var-dec Ba)
                              :: vArg (var-dec (var-dec (sources₀ Cab))) :: []))
                              ↪  :: [])
sources₁ (Π[ x : arg i A ] Bx) = A
sources₁ (def (quote Σ) (ℓ₁ :: ℓ₂ :: τ :: body))
    = def (quote Σ) (ℓ₁ :: ℓ₂ :: map-Arg sources₀ τ :: List.map (map-Arg
    ↪  sources₁) body)
-- This function introduces 𝟙s, so let's drop any old occurances a la 𝟘.
sources₁ (def (quote ⊤) _) = def (quote 𝟘) []
sources₁ (lam v (abs s x))      = lam v (abs s (sources₁ x))
sources₁ (var x args) = var x (List.map (map-Arg sources₁) args)
sources₁ (con c args) = con c (List.map (map-Arg sources₁) args)
sources₁ (def f args) = def f (List.map (map-Arg sources₁) args)
sources₁ (pat-lam cs args) = pat-lam cs (List.map (map-Arg sources₁) args)
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources₁ t = t


macro
  sources : Term → Term → TC Unit.⊤
  sources tm goal = normalise tm >>=ₘ λ tm′ → unify (sources₁ tm′) goal
```

## Stage 4: Σ→⊎ –Replacing Products with Sums

As another tersely introduced utility, let us flesh-out Σ→⊎ by means of a few desired unit tests —notice that the final example concerns a parameterised dynamical system. As mentioned in the guiding calculation, we will replace unit types by empty types —i.e., "empty Σ-products by empty ⊎-sums".

| τ | Σ→⊎ τ |
|---|---|
| `Π S : Set • (S → S)` | `Π S : Set • (S → S)` |
| `Π S : Set • Σ n : S • S` | `Π S : Set • S ⊎ S)` |
| `Π S : Set • Σ n : (S → S) • S` | `Π S : Set • (S → S) ⊎ S)` |
| `Π S : Set • Σ s : S • Σ n : (S → S) • 𝟙` | `Π S : Set • S ⊎ (S → S) ⊎ 𝟘` |

**Decreasing de Brujin Indices:** Any given quantification `(Σ x : τ • fx)` may have its body `fx` refer to the free variable `x`. If we decrement all de Bruijn indices `fx` contains, then there would be no reference to `x`. ( In the repository code, ↓↓ appears as `var-dec`. )

↓↓

Building up to the **termtype** combinator

```
↓↓₀ : (fuel : ℕ) → Term → Term
↓↓₀ zero t  = t
-- Let's use an "impossible" term.
↓↓₀ (suc n) (var zero args)      = def (quote ⊥) []
↓↓₀ (suc n) (var (suc x) args)   = var x args
↓↓₀ (suc n) (con c args)         = con c (map-Args (↓↓₀ n) args)
↓↓₀ (suc n) (def f args)         = def f (map-Args (↓↓₀ n) args)
↓↓₀ (suc n) (lam v (abs s x))    = lam v (abs s (↓↓₀ n x))
↓↓₀ (suc n) (pat-lam cs args)    = pat-lam cs (map-Args (↓↓₀ n) args)
↓↓₀ (suc n) (Π[ s : arg i A ] B) = Π[ s : arg i (↓↓₀ n A) ] ↓↓₀ n B
{-# CATCHALL #-}
-- sort, lit, meta, unknown
↓↓₀ n t = t


↓↓ : Term → Term
↓↓ t = ↓↓₀ (lengthₜ t) t
```

Notice that we made the decision that x, in the body of `(Σ x • x)`, will reduce to 𝟘, the empty type. Indeed, in such a situation the only Debrujin index cannot be reduced further; e.g., `↓↓(quoteTerm x) ≡ quoteTerm ⊥`.

Σ→⊎

```
↓↓ τ = “reduce all de-bruijn indices within τ by 1”
Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (↓↓ Ba)
```

Building up to the `termtype` combinator

```
{-# TERMINATING #-}
Σ→⊎₀ : Term → Term
Σ→⊎₀ (def (quote Σ) (h₁ :: h₀ :: arg i A :: arg i₁ (lam v (abs s x)) :: []))
  = def (quote _⊎_) (h₁ :: h₀ :: arg i A :: vArg (Σ→⊎₀ (↓↓ x)) :: [])
-- Interpret “End” in do-notation to be an empty, impossible, constructor.
Σ→⊎₀ (def (quote ⊤) _) = def (quote ⊥) []
 -- Walk under λ's and Π's.
Σ→⊎₀ (lam v (abs s x)) = lam v (abs s (Σ→⊎₀ x))
Σ→⊎₀ (Π[ x : A ] Bx) = Π[ x : A ] Σ→⊎₀ Bx
{-# CATCHALL #-}
Σ→⊎₀ t = t

macro
  Σ→⊎ : Term → Term → TC Unit.⊤
  Σ→⊎ tm goal = normalise tm >>=ₘ λ tm′ → unify (Σ→⊎₀ tm′) goal
```

We can now form the fourth stage approximation of the functor $\mathcal{D}$; in-fact we will use this form as *the definition* of the desired functor $\mathcal{D}$ —since the sum with 𝟘 *essentially* contributes nothing.

Building up to the `termtype` combinator

```
D₄ = Σ→⊎ D₃

4-unions : D₄ ≡ λ X → 𝟙 ⊎ X ⊎ 𝟘
4-unions = refl
```

## Stage 5: Fixpoint

Since we want to define algebraic data-types as fixed-points, we are led inexorably to using a recursive type that fails to be positive.

Building up to the termtype combinator

```
{-# NO_POSITIVITY_CHECK #-}
data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
  μ : F (Fix F) → Fix F

𝔻 = Fix D₄
```

We summarise the stages together into one macro:

Termtype

```
termtype : UnaryFunctor → Type
termtype τ = Fix (Σ→⊎ (sources τ))
```

Building up to the termtype combinator

```
macro
  termtype : Term → Term → TC Unit.⊤
  termtype tm goal =
                normalise tm
           >>=ₘ λ tm′ → unify goal (def (quote Fix) ((vArg (Σ→⊎₀ (sources₁
           ↪  tm′))) :: []))
```

Then, we may instead declare:

Building up to the termtype combinator

```
𝔻 = termtype (DynamicSystem :waist 1)
```

### 5.3.2 Instructive Example: $\mathbb{D} \cong \mathbb{N}$

It is instructive to work through the process of how $\mathbb{D}$ is obtained from `termtype` in order to demonstrate that this approach to algebraic data types is practical **within Agda**.

Declaring a Derived Termtype

```
𝔻 = termtype (DynamicSystem :waist 1)

-- Pattern synonyms for more compact presentation
pattern startD  = μ (inj₁ tt)        -- : 𝔻
pattern nextD e = μ (inj₂ (inj₁ e)) -- : 𝔻 → 𝔻
```

With these `pattern` declarations, we can actually use the more meaningful names **startD** and **nextD** when pattern matching, instead of the seemingly daunting $\mu$-**inj**-ections. For instance, we can immediately see that the natural numbers act as the description language for dynamical systems:

Seemingly Trivial Remappings

```
to : 𝔻 → ℕ
to startD    = 0
to (nextD x) = suc (to x)

from : ℕ → 𝔻
from zero    = startD
from (suc n) = nextD (from n)
```

Readers whose language does not have `pattern` clauses need not despair. With the following macro

$$\texttt{Inj n x} = \mu\ (\texttt{inj}_2\ \boxed{^n}\ (\texttt{inj}_1\ \texttt{x}))$$

Seemingly Trivial Remappings

```
-- i-th injection:  (inj₂ ∘ ··· ∘ inj₂) ∘ inj₁
Inj₀ : ℕ → Term → Term
Inj₀ zero c    = con (quote inj₁) (arg (arg-info visible relevant) c ∷ [])
Inj₀ (suc n) c = con (quote inj₂) (vArg (Inj₀ n c) ∷ [])

macro
  Inj : ℕ → Term → Term → TC Unit.⊤
  Inj n t goal = unify goal ((con (quote μ) []) app (Inj₀ n t))
```

we may define `startD = Inj 0 tt` and `nextD e = Inj 1 e`—that is, constructors of termtypes are particular injections into the possible summands that the termtype consists of.

## 5.4 Free Datatypes from Theories

Astonishingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if a parameterised context `𝒞 : Set → Context ℓ₀` is given, then `ℂ = λ X` $\boxed{\to}$ `termtype (𝒞 X :waist 1)` can be used to form 'free, lawless, $\mathcal{C}$-instances'. For instance, earlier we witnessed that the termtype of dynamical systems is essentially the natural numbers.

Data structures as free theories

| Theory | Termtype |
|---|---|
| Dynamical Systems | ℕ |
| Pointed Structures | Maybe |
| Monoids | Binary Trees |

The final entry in the above table is a well known correspondence that we can now not only formally express, but also prove to be true. As we did with dynamical systems, we begin with forming 𝕄 the termtype of monoids, then using `pattern` clauses to provide compact names, and explicitly form the algebraic `data` type of trees.

Trees from Monoids

```
𝕄 : Set
𝕄 = termtype (Monoid ℓ₀ :waist 1)
{- i.e., Fix (λ X → 𝟙       -- Id, nil leaf
                ⊎ X × X × 𝟙 -- _⊕_, branch
                ⊎ 𝟘         -- invariant leftId
                ⊎ 𝟘         -- invariant rightId
                ⊎ X × X × 𝟘 -- invariant assoc
                ⊎ 𝟘)        -- the “End {ℓ}”
-}

-- Pattern synonyms for more compact presentation
pattern emptyM      = μ (inj₂ (inj₁ tt))              -- : 𝕄
pattern branchM l r = μ (inj₁ (l , r , tt))           -- : 𝕄 → 𝕄 → 𝕄
pattern absurdM a   = μ (inj₂ (inj₂ (inj₂ (inj₂ a)))) -- absurd 𝟘-values

data TreeSkeleton : Set where
  empty  : TreeSkeleton
  branch : TreeSkeleton → TreeSkeleton → TreeSkeleton
```

Using Agda's Emacs interface, we may interactively case-split on values of 𝕄 until the declared patterns appear, then we associate them with the constructors of `TreeSkeleton`.

Seemingly Trivial Remappings

```
to : 𝕄 → TreeSkeleton
to emptyM        = empty
to (branchM l r) = branch (to l) (to r)
to (absurdM (inj₁ ()))
to (absurdM (inj₂ ()))

from : TreeSkeleton → 𝕄
from empty        = emptyM
from (branch l r) = branchM (from l) (from r)
```

That these two operations are inverses is easily demonstrated.

Trees from Monoids

```
fromoto : ∀ m → from (to m) ≡ m
fromoto emptyM        = refl
fromoto (branchM l r) = cong₂ branchM (fromoto l) (fromoto r)
fromoto (absurdM (inj₁ ()))
fromoto (absurdM (inj₂ ()))

toofrom : ∀ t → to (from t) ≡ t
toofrom empty        = refl
toofrom (branch l r) = cong₂ branch (toofrom l) (toofrom r)
```

Without the `pattern` declarations the result would remain true, but it would be quite difficult to believe in the correspondence without a machine-checked proof.

To obtain a data structure over some 'value type' Ξ, one must start with "theories containing a given set Ξ". For example, we could begin with the theory of abstract collections, then obtain lists as the associated termtype.

Lists from Paramterised Collections

```
Collection : ∀ ℓ → Context (ℓsuc ℓ)
Collection ℓ = do Elem    ← Set ℓ
                  Carrier ← Set ℓ
                  insert  ← (Elem → Carrier → Carrier)
                  ∅       ← Carrier
                  End {ℓ}

ℂ : Set → Set
ℂ Elem = termtype ((Collection ℓ₀ :waist 2) Elem)

pattern _::_ x xs = μ (inj₁ (x , xs , tt))
pattern  ∅        = μ (inj₂ (inj₁ tt))
```

Realising Collection ASTs as Lists

```
to : ∀ {E} → ℂ E → List E
to (e :: es) = e ∷ to es
to ∅         = []
```

It is then little trouble to show that `to` is invertible. We invite the readers to join in on the fun and try it out themselves.

## 5.5 Related Works

Surprisingly, conflating parameterised and non-parameterised record types with termtypes *within a language in a practical fashion* has not been done before.

The `PackageFormer` **DBLP:conf/gpce/Al-hassyCK19**; **alhassy_thesis_proposal** editor extension reads contexts —in nearly the same notation as `Context`— enclosed in dedicated comments, then generates and imports Agda code from them seamlessly in the background whenever typechecking happens. The framework provides a fixed number of meta-primitives for producing arbitrary notions of grouping mechanisms, and allows arbitrary Emacs Lisp **10.5555/229872** to be invoked in the construction of complex grouping mechanisms.

Comparing the in-language Context mechanism with the PackageFormer editor extension

| | PackageFormer | Contexts |
|---|---|---|
| Type of Entity | Preprocessing Tool | Language Library |
| Specification Language | Lisp + Agda | Agda |
| Well-formedness Checking | × | ✓ |
| Termination Checking | ✓ | ✓ |
| Elaboration Tooltips | ✓ | × |
| Rapid Prototyping | ✓ | ✓ (Slower) |
| Usability Barrier | None | None |
| Extensibility Barrier | Lisp | Weak Metaprogramming |

The previous chapter provided the syntax of `PackageFormer` necessary to form useful grouping mechanisms but was shy on the semantics of such constructs. We have chosen the names of the `Context` combinators to closely match those of `PackageFormer`'s with an aim of furnishing the mechanism with semantics by construing the syntax as semantics-functions; i.e., we have a shallow embedding of `PackageFormer`'s constructs as Agda entities:

Context as a semantics for PackageFormer constructs

| Syntax | Semantics |
|---|---|
| `PackageFormer` | `Context` |
| `:waist` | `:waist` |
| ⟴ | Forward function application |
| `:kind` | `:kind`, see below |
| `:level` | Agda built-in |
| `:alter-elements` | Agda macros |

`PackageFormer`'s `_:kind_` meta-primitive dictates how an abstract grouping mechanism should be viewed in terms of existing Agda syntax. However, unlike `PackageFormer`, **all**

**of our syntax consists of legitimate Agda terms.** Since language syntax is being manipulated, we are forced to implement the `_:kind_` meta-primitive as a macro.

Kind

```
𝒞 :kind ‘record    = 𝒞 0
𝒞 :kind ‘typeclass = 𝒞 :waist 1
𝒞 :kind ‘data      = termtype (𝒞 :waist 1)
```

Codes for Agda's first-class grouping mechanisms

```
data Kind : Set where
  ‘record     : Kind
  ‘typeclass  : Kind
  ‘data       : Kind

macro
  _:kind_ : Term → Term → Term → TC Unit.⊤
  _:kind_ t (con (quote ‘record) _)    goal
      = normalise (t app (quoteTerm 0))
        >>=ₘ λ t′ → unify (waist-helper 0 t′) goal
  _:kind_ t (con (quote ‘typeclass) _) goal
      = normalise (t app (quoteTerm 1))
        >>=ₘ λ t′ → unify (waist-helper 1 t′) goal
  _:kind_ t (con (quote ‘data) _) goal
      = normalise (t app (quoteTerm 1))
        >>=ₘ λ t′ → normalise (waist-helper 1 t′)
        >>=ₘ λ tt → unify goal (def (quote Fix)
                                  ((vArg (Σ→⊎₀ (sources₁ tt))) ∷ []))
  _:kind_ t _ goal = unify t goal
```

We did not expect to be able to define a full Agda implementation of the semantics of `PackageFormer`'s syntactic constructs due to Agda's rather constrained metaprogramming mechanism. However, it is important to note that `PackageFormer`'s Lisp extensibility expedites the process of trying out arbitrary grouping mechanisms —such as partial-choices of pushouts and pullbacks along user-provided assignment functions— since it is all either string or symbolic list manipulation. On the Agda side, using `Context`, it would require substantially more effort due to the limited reflection mechanism and the intrusion of the stringent type system.

## 5.6 Conclusion

Starting from the insight that related grouping mechanisms could be unified, we showed how **related structures can be obtained from a single declaration using a practical**

**interface.** The resulting framework, based on contexts, still captures the familiar record declaration syntax as well as the expressivity of usual algebraic datatype declarations —at the minimal cost of using `pattern` declarations to aide as user-chosen constructor names. We believe that our approach to using contexts as general grouping mechanisms *with* a practical interface are interesting contributions.

We used the focus on practicality to guide the design of our context interface, and provided interpretations both for the rather intuitive "contexts are name-type records" view, and for the novel "contexts are fixed-points" view for termtypes. In addition, to obtain parameterised variants, we needed to explicitly form "contexts whose contents are over a given ambient context" —e.g., contexts of vector spaces are usually discussed with the understanding that there is a context of fields that can be referenced— which we did using the name binding machanism of `do`-notation. These relationships are summarised in the following table.

Contexts embody all kinds of grouping mechanisms

| Concept | Concrete Syntax | Description |
|---|---|---|
| Context | `do S ← Set; s ← S; n ← (S → S); End` | "name-type pairs" |
| Record Type | `Σ S : Set • Σ s : S • Σ n : S → S • 𝟙` | "bundled-up data" |
| Function Type | `Π S • Σ s : S • Σ n : S → S • 𝟙` | "a type of functions" |
| Type constructor | `λ S • Σ s : S • Σ n : S → S • 𝟙` | "a function on types" |
| Algebraic datatype | `data 𝔻 : Set where s : 𝔻; n : 𝔻 → 𝔻` | "a descriptive syntax" |

To those interested in exotic ways to group data together —such as, mechanically deriving product types and homomorphism types of theories— we offer an interface that is extensible using Agda's reflection mechanism. In comparison with, for example, special-purpose preprocessing tools, this has obvious advantages in accessibility and semantics.

To Agda programmers, this offers a standard interface for grouping mechanisms that had been sorely missing, with an interface that is so familiar that there would be little barrier to its use. In particular, as we have shown, it acts as **an in-language library for exploiting relationships between free theories and data structures.** As we have presented the high-level definitions of the core combinators —alongside Agda-specific details which may be safely ignored— it is also straightforward to translate the library into other dependently-typed languages.

# 6 Conclusion

The initial goal of this work was to explore how investigations into packaging-up-data —and language extension in general— could benefit from mechanising tedious patterns, thereby reinvigorating the position of universal algebra within computing. Towards that goal, we have decided to create an editor extension that can be used, for instance, to quickly introduce universal algebra constructions for the purposes of "getting things done" in a way that does not force users of an interface to depend on features they do not care about —the so-called Interface Segregation Principle. Moreover, we have repositioned the prototype from being an auxiliary editor extension to instead being an in-language library and have presented its key insights so that can be developed in other dependently-typed settings besides Agda.

Based on the results —such as the 750% line savings in the MathScheme library— we are convinced that the (one-line) specification of common theories (data-structures) can indeed be used to reinvigorate the position of universal algebra in computing, as far as DTLs are concerned. The focus on the modular nature of algebraic structures, for example, allows for the *mechanical* construction of novel and unexpected structures in a practical and elegant way —for instance, using the `keeping` combinator to extract the *minimal* interface for an operation, or proof, to be valid. Also, we believe that the correspondence between abstract mathematical theories and data structures in computing only strengthens the need for a mechanised approach for the under-utilised constructions available on the the mathematical side of the correspondence.

Some preliminary experiences show that the approach used in this thesis can be used with immediate success. For example, the editor extension allows a host of renamings to be done, along with the relevant relationship mappings, and so allow proofs to be written in a more readable fashion. As another example, the in-language library allows one to show that the free algebra associated with a theory is a particular useful and practical data-structure —such as $\mathbb{N}$, `Maybe`, and `List`. These two examples are more than encouraging, for the continual of this effort. Also, the success claimed by related work like [**Arend**] and [**that-group**] makes us believe that we can have a positive impact.

This thesis has focused on various aspects of furnishing packages with a status resemebling that of a first-class citizen in a dependently-typed language. Where possible, we will give an indication of future work which has still to be done to get more insight in this direction.

## 6.1 Questions, Old and New

Herein we revisit the research questions posed in the introductory chapter, summarise our solutions to each, and discuss future work.

**Practical Concern ♯1: Renaming & Remembering Relationships.** A given structure may naturally give rise to various 'children structures', such as by adding-new/dropping-old/renaming componenets, and it is useful to have a (possibly non-symmetric) coercision between the child and the original parent.

We have succeeded to demonstrate that ubquitious constructions can be mechanised and the coercisions can also be requested by a simple keyword in the specification of the child structure. As far as this particular problem is concerned, we see no missing feature and are content with the success that the PackageFormer prototype has achieved. However, the in-language Context library does leave room for improvement, but this is a limitation of the current Agda reflection mechanism rather than of the approach outlined by PackageFormer.

**Practical Concern ♯2: Unbundling.** A given structure may need to have some of its components 'fixed ahead of time'. For instance, if we have a type `Graph` of graphs but we happen to be discussing only graphs with natural numbers as nodes, then we need to work with `Σ G : Graph • G.Node ≡ ℕ` and so work with pairs `(G, refl)` whose second component is a necessarily technical burden, but is otherwise insightful.

Our framework(s) achieve this goal, joyously so. An improvement would be not to blindly lift the first $n$-many componenets to the type level but instead to expose the induced depenencdy subgraph of a given set of componenets. The PackageFormer already does this for the `keeping` combinator and the same code could be altered for the `waist` combinator. At first, it would seem that a similar idea would work for the in-language library, however this is not the case. The Context library, unlike PackageFormer, does not work with flat strings but instead transforms the inner nodes of abstract syntax trees —such as replacing Πs by λs or Σs— and so the need to lift a subgraph of a structure's signature no longer becomes a linear operation that alters inner nodes.

Perhaps an example would illuminate the problem. Consider the following signature.

PSGwId$^2$ —'P'ointed 'S'emi'g'roup 'w'ith 'Id$^2$' ≈ Id

```
record PSGwId² : Set₁ where
  field
    -- We have a semigroup
    C     : Set
    _⊕_    : C → C → C
    assoc : ∀ x y z  →  (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
    -- with a selected point
    id    : C

  twice : C → C
  twice = λ x → x ⊕ x

  -- Such that the point is idempotent
  field
    id² : twice id ≡ id
```

Suppose we want to have the field `id`$^2$ at the type level, then we must also expose the parts of the signature that make it well-defined; namely, `C, _⊕_, id, twice`. At a first pass, `id`$^2$ only needs `id` and the operation `twice`; however, if we look at each of these in-turn we see that we also need `C` and `_⊕_`. As such, in the worst case, this operation is quadratic. Moving on, as the signature is traversed, we can mark fields to be lifted but we need a combinator to "shift leftward (upward)" the names that are to be at the type level —in this case, we need to move `id`$^2$ and `id` to come before `assoc`. This is essentially the algorithm implemented in PackageFormer's `keeping` combinator. However, for Context's `do`-notation, this may not be possible since inner-nodes are no longer replaced, linearly, according to a single toggle. Furture work would be to investigate whether it would be possible and, if so, how to do so in a *pragmattic and usable* fashion.

**Theoretical Concern ♯1: Exceptionality.** If an integer $m$ divides an integer $n$, then division $nm$ yields an integer witnessing $n$ as a multiple of $m$; likewise, if a package $p$ is structurally (nominally) contained in a package $q$, then we can form a pacakge, say, $q - p$ that contains the extra matter and it is parameterised by an instance of $p$ —e.g., `Monoid` is contained in `Group` and so `Group - Monoid = λ (M : Monoid) → (_⁻¹ : ···, left-inverse : ···, right-inverse: ···)` is the paramterised pacakage that can adjoin inverses to monoids. As such, packages are like numbers —compare with the idea that a list is like a number, the latter being a list of unit (trivial) information.

Our goal was to determined the *feasiblility* of this idea *within* dependently-typed settings. The implementation of the the Context in-language library yields a resounding positive. As mentioned already, limitations of the host DTL's reflection mechanism are inherited by our approach.

Furture work would focus on the precise relationship between features of the host language and a library treating packages as first-class. Moreover, it would be useful to investigate how packages can be promoted to first-class *after* the construction of a language. Such an investigation would bring to light the interplay of how packages actually influence other parts of a language —which is sorely lacking from our work.

Perhaps the most pressing concern would be how the promotion of packages would influence typechecking. At first, for instance, the package `PSGwId`[2] from above could be typed as $\texttt{Set}_1$ but that would be wildely inappropriate since we cannot apply arbitrary package combinators, such as `_∸_`, to arbitrary types —just as we cannot apply `_÷_` to arbitrary types. Instead, we would need a dedicated type, say, `Package`. Things now become exceedingly hairy. Do we need a hierarchy or avoid paradoxes, as is the case with $\texttt{Set}_n$? A parameterised type is a Π-type, but a paramterised package *is a* package —so do Π-types get 'absorped' into `Package`? What are the types of the package combinators introduced in this thesis, such as unbundliungs $\Pi{\to}\lambda$?

These questions are not only interesting by themselves but we also would be a stepping stone in having full-fledged first-class pacakages in dependently-typed languages.

**Theoretical Concern ♯2: Syntax.** The theories-as-data-structures lens presented in this work showcases how a theory (a record type, signature, admitting instances) can have useful data-structures (algebraic data types) associated with it. For instance, monoids give rise to binary trees whose leaf values are drawn from a given carrier (variable) set. One can then encode a sentence of a model structure using the syntax, perform a syntactic optimisation, then interpret the sentence using the given instance.

We are delighted with the rather unexpected success of this aspect of our work. The formal methods community is well-aware that monoids are related to binary trees and that pointed sets are related to maybe (nullable) types, yet we have had the honour of being the first to actually derive the latter from the former *mechanically.*

Future work would focus on the treatment non-function-symbols. For instance, instead of discarding properties from a theory, one could keep them thereby obtaining 'higher-order datatypes' [**cubical_agda**] or could have them lifted as parameters in a (mechanically generated) subsequent module. Moreover, the current implementation of Context has a basic predicate determining what consitutes a function-symbol, it would be interesting to make that a parameter of the theories-as-data-structures `termtype` construction.

**Proof.** Finally, there are essentially no formal theorems proven in this work. The constructions presented rely on *typechecking*: One can phrase a desired construction and typechecking determines whether it is meaningful or not. It would be useful to determine the necessary conditions that guarantee the well-definedness of the constructions —so that we may then "go up another level" and produce meta-constructions that invoke our current constructions mechanically and "wholesale".

## 6.2 Concluding Remarks

In dependently-typed settings (DTS), it is common practice to operate on packages —by renaming them, hiding parts, adding new parts, etc.— and the frameworks presented in this thesis show that it is indeed possible to treat packages nearly as first-class citizens "after the fact" even when a language does not assign them such a status. The techniques presented show that this approach is feasible as an in-language library for DTS as well as for the any highly customisable and extensible text editor.

The combinators presented in this thesis were guided not by theoretical concerns on the algebraic nature of containers but rather on the practical needs of actual users working in DTS. We legitimately believe that that our stance on packages as first-class citizens should —and hopefully one day would— be an integral part of any DTS. The Context library is a promising approach to promoting the status of packages, to reducing the gap between different "sub-languages" in a language, and allowing users to benefit from a streamlined and familiar approach to packages —as if they were the 'fancy numbers' abstracted by rings, fields, and vector spaces.

Finally, even though we personally believe in the import of packages, we do not expect the same belief to trickle-down to mainstream languages immediately since they usually do not have sufficiently sophisticated[1] type systems to permit the treatment of packages as first-class citizens, on the same footing as numbers. Nonetheless, we believe that the work in this thesis is yet another stepping-stone on the road of *DRY*[2] endeavours.

[1] The static typing of some languages, such as C, is so pitiful that is makes type systems seem more like a burden then anything useful —in C, one often uses void pointers to side-step the type system's limitations, thereby essentially going untyped. The dynamically typed languages, however, could be an immediate test-bed for package combinators —indeed, Lisp, Python, and JavaScript use 'splicing' operators to wholesale include structures in other structures, within the core language.

[2] *Don't Repeat Yourself!*