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

November 11, 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., Π-, Σ-, 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.

A middle-path with margins

Imagine having to stop reading mid-sentence, go to the bottom of the page, read a footnote, then stumble around till you get back to where you were reading[0] . Even worse is when one seeks a cryptic abbreviation and must decode a world-away, in the references at the end of the document.

[0]No more such oppression! Consequently, we reset sidenote counters at the start of each chapter.

I would like you to be able to read this work *smoothly, with minimal interpretations.* As such, inspired by [11] among others, we have opted to include "mathematical graffiti" in the margins. In particular, the margins side notes may have *informal and optioniated* remarks[β]. We're trying to avoid being too dry, and aim at being somewhat light-hearted.

[11] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994. ISBN: 0-201-55802-5. URL: `https://www-cs-faculty.stanford.edu/%5C%7Eknuth/gkp.html`

Dijkstra [5] might construe the graffiti as *mathematical politeness* that could potentially save the reader a minute. Even though a characteristic of academic writing is its terseness[ω], we don't want to baffle or puzzle our readers, and so we use the informality of the graffiti to say what we mean bluntly, *but* it may be less accurate or not as formally justifiable as the text proper.

β Professional academic writing to the left; here in the right we take a relaxed tone.

[5] Edsger W. Dijkstra. *The notational conventions I adopted, and why.* circulated privately. July 2000. URL: `http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF`

ω "It's so obvious, I won't waste time on it"; i.e., "It's an exercise to the reader to figure out what I'm really saying." Elaboration removes mystery and some authors might prefer academia be exclusive.

> *Some consider the puzzles that are created by their omissions as spicy challenges, without which their texts would be boring; others shun clarity lest their worth is considered trivial. [...] Some authors believe that, in order to keep the reader awake, one has to tickle him with surprises. [...] essential for earning the respect of their readership.*
> *—Edsger Dijkstra [5]*

When there are no side remarks to be made, or a code snippet would be better viewed with greater width, we will unabashedly switch to using the full width of the page —temporarily, on the fly, and without ceremony.

In particular, in numerous places, we want to show the *exact* code generated from our prototype —rather than an after-the-fact prettification, which would undermine the 'utility' of the tool.

A superficial cost of utilising margin space is that the overall page count may be 'over-exaggerated'[γ]. Nonetheless, I have found long empty columns of margin space *yearning* to be filled with explanatory remarks, references, or somewhat helpful diagrams. Paraphrasing Hofstadter [14], the little pearls in the margins were so connected in my own mind with the ideas that I was writing about that for me to deprive my readers of the connection that I myself felt so strongly would be nothing less than perverse.

γ Which doesn't matter, since you're likely reading this online!

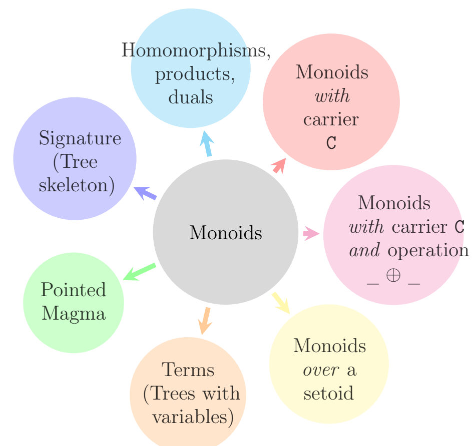
[14] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books Inc., 1979

Contents

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*. The figure to the right 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.

Deriving related *types* from *the* definition of monoids



When developing libraries, such as [17], in the dependently-typed language (DTL) Agda, one is forced to mitigate a number of hurdles. We turn to these hurdles in the following subsections —some of which are also discussed clearly in [3]. The remainder of this chapter is organised as follows: Sections 1.1 to 1.4 discussing the motivating problems[0] that arise when working in a DTL, then Section 1.5 briefly discusses our desire to have our resulting system be *usable*, and, finally, Section 1.6 concludes with an overview of the thesis as well as providing an estimate of the accessibility —interdependence— of the remaining chapters.

[17] Wolfram Kahl. *Relation-Algebraic Theories in Agda*. 2018. URL: `http://relmics.mcmaster.ca/RATH-Agda/` (visited on 10/12/2018)

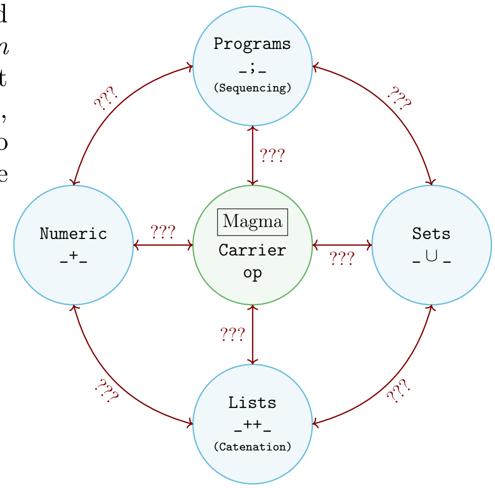
[3] Jacques Carette and Russell O'Connor. "Theory Presentation Combinators". In: *Intelligent Computer Mathematics* (2012), pp. 202–215. DOI: `10.1007/978-3-642-31374-5_14`

[0] Discussed in greater detail in Chapter 3.

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 to the right, 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' `coe`rcions have to be written by hand.

Given green, derive cyan candidate constructions, require red relationships




```
coe : Numeric → Magma
coe record {Numeric = N; _+_ = op}
 = record {Carrier = N; op = op}
```

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** [8]. Unfortunately, library developers generally provide only a few *variations* on *a* package; such as having no parameters or having only *functional symbols* as parameters[1] . 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.

[8] François Garillot et al. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. Ed. by Tobias Nipkow and Christian Urban. Vol. 5674. Lecture Notes in Computer Science. Munich, Germany: Springer, 2009. URL: https://hal.inria.fr/inria-00368403

[1] Recall the carrier C and operation $\_\oplus\_$ in the above figure on monoid constructions.

Let us try to 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 symbol '≅' means "isomorphic with" and it means "essentially interchangeable". More formally, it signals that there is a non-lossy protocol between two types. It is most generally defined in the setting of Category Theory: $A \cong B$ *precisely* when there are two transformations $f : A \to B$ and $g : B \to A$ that 'undo one another' in that $f \circ g = \mathsf{Id} = g \circ f$.

The right side brings a number of *practical conveniences* in the form of simplified concrete syntax —e.g., reduced parentheses 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):

Notice that *before* I, X, Y were *independent* types; whereas *here* we have that Y depends on I and X, and X depends on I.

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

Dependent types and type-formers such as 'Σ' and 'Π' are defined in chapter 2.

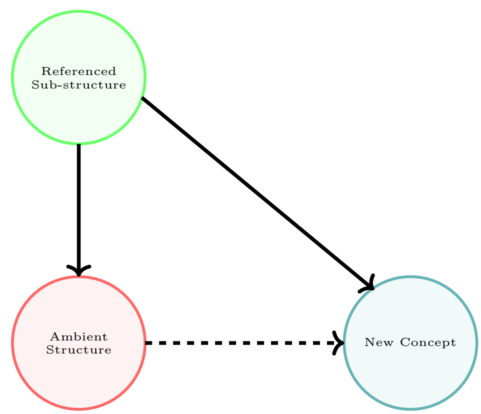
As with currying, the right side here is preferable at times since it immediately lets one 'fix' —i.e., select— a value $i_0 : I$ to obtain the specialised type

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

In contrast to the right, the left side can only be contorted to simulate the idea of fixing a field, $i_1 : I$, ahead of time; e.g.:

$$\Pi p : (\Sigma i : I \bullet X\, i) \bullet Z\, p \quad \text{where} \quad Z\, p \;=\; \Big(Y\, p \times (\mathsf{fst}\, p \equiv i_1)\Big)$$

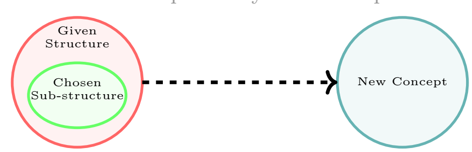
The verbosity of this formulation is what we wish to mitigate.



Bundled forms: Two solid arrows to get one dashed arrow

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 'Σ'. 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; as shown in the informal[2] 3-node diagram to the right. 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'; as shown in the 2-node diagram to the right.

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



Unbundled forms: Obtain the dashed arrow explicitly

It is interesting to note that the unbundling problem appears in a number of guises within the setting of programming language design. For instance, it can be seen in numerous popular languages, including Haskell and JavaScript, in the form of *pattern matching*, or *de-structuring*; wherein **explicit** treatment of record arguments as *packaging mechanisms,* **silently** disspears in the *presentation* of function definitions. Then, *implicit curriying* is the feature that allows the presentation to accomodated arguments *sequentially* ("one at a time") rather than "all at once". The move from function-formation 'λ' to type-formation 'Π' results in essentially the so-called *quantifier nesting* rules of predicate logic [12].

Define `f : X × Y → Z`
by projecting fields as needed
`f p = ··· fst p ··· snd p ···`
or by exposing the fields directly
`f (x, y) = ··· x ··· y ···.`
But to 'curry' is another matter:
`f′ = λ x • λ y • ··· x ··· y ···.`

[12] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer, 1993. ISBN: 0-387-94115-0. DOI: 10.1007/978-1-4757-3837-7. URL: https://doi.org/10.1007/978-1-4757-3837-7

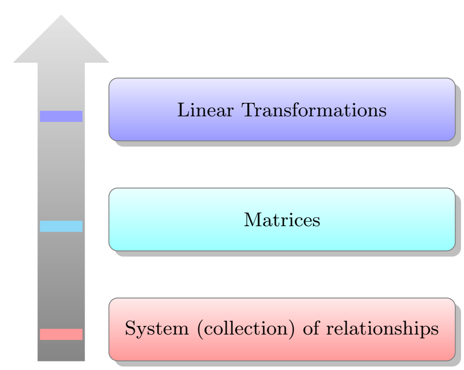
1.3 Theoretical Concern ♯1: Exceptionality

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[3] . Unfortunately[4] , 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.

[3]For example, programs and proofs are essentially the same thing. This is known as the *Curry-Howard Correspondence* and as the *Types-as-Propositions Correspondence.*

[4]There are rare exceptions. E.g., some members of the non-DTL ML language family allow first-class modules.

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* (expressing equations[5]) 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.



[5]The matrix equation $A \cdot x = B$ captures the system of equations with coefficients from A, unknowns from x, and B are the 'target coefficients'.

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 parameterised 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.

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
  Leaf   : C → Tree C
  Branch : Tree C
               → Tree C → Tree C
  Nil    : Tree C
```

Hint ...

`_⨾_`	$\approx$	`Branch`
`Id`	$\approx$	`Nil`

For example, as shown in the first diagram of the thesis, the record type of monoids models composition, whereas the termtype of binary trees acts as a description language for monoids. These can be rendered in Agda, as shown above. The ***problem of maintenance*** now arises: Whenever the record type is altered, one must mechanically update the associated termtype.

"Termtype?"

We will refer to algebraic data types as *termtypes*, rather than *term type* nor *term-type*. The reason for doing so is that in Chapter 2 we will discuss *terms* and *types*, and come to see them as indistinguishable —for the most part. As such, the phrase *term type* could be read ambiguously as "the type of terms" or as "the term denoting a type". For these reasons, we have chosen "termtype". Moreover, in Chapter 5, we will form a macro that consumes a particular kind of package and yields a termtype: The name of the macro is `termtype`.

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; i.e., selected, automatic, protocals— 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**.

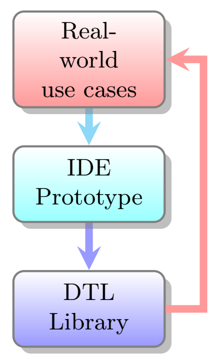
If you can't use it, it's essentially useless!

A concrete example is demonstrated later on, such as in Figure ??.

1.6 Thesis Overview

The remainder of the thesis is organised as follows.

"Thesis guidline"!



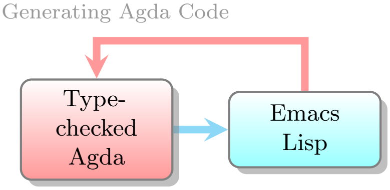
Chapter 2 consists of preliminaries, to make the thesis self-contained, and lists the contributions of the thesis.

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.

Generating Agda Code



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".

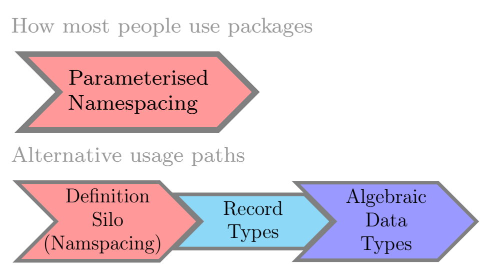
Chapter 5 takes the lessons learned from the prototype to show that *DTLs can have a unified package system within the host language.*

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 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.



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

—Not yet re-worked into this new marginful format—

3 Motivating the problem —Examples from the Wild

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 [0] .

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

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"[1] 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 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.

[0] 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 ??.

[1] 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**.

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. We begin with a ubqutious problem[2] that happens in practice: Given a list $[x_0,\ x_1,\ \ldots,\ x_{n-1}]$, 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.

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".

[2]A variation of this problem is discussed in section ??.

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 $[x_0,\ x_1,\ \ldots,\ x_{n-1}]$ is composed by repeatedly prepending new elements to the front of existing lists, starting from an empty list. That is, the informal notation $[x_0,\ x_1,\ \ldots,\ x_{n-1}]$ 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
```

Then, to define `head l` for any list `l`, we consider the *possible shapes* of the variable 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[3] function or one has to "add fictitious elements to every type"[4] such as $\mathsf{undefined}_A : A$. However, in a DTL, 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.*

Trying to define the `head` function.

Partially defined `head`

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

[3]Leaving users the burden of ensuring that any call `head l` never happens with `l = []`! Otherwise, we need to parameterise our function by a "default value".

[4]Thereby having no empty types at all —roughly put, this is what Haskell does. Agda lets us do this with the `postulate` keyword.

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[5] 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 positive length and so is non-empty; i.e., non-emptiness can be expressed directly in the type of `l`.

[5] The definition of this type, and the subsequent `head` function, have been discussed in section ??, 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[6] —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[7] 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

[6] 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.

[7] 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.

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.

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.

Equivalent structures, but different usability profiles.

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
```

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. But for such an 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.

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′)`.

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.

For brevity, rather than consider program language phrases and operators on them, we abstract to bi-magmas —which will be seen again in Chapter 4!

To bundle or to not bundle?

```
record Magma0 : Set1 where
  constructor ⟨_,_⟩0
  field
    Carrier : Set
    _⨾_ : Carrier → Carrier → Carrier

record Magma1 (Carrier : Set) : Set1 where
  constructor ⟨_⟩1
  field
    _⨾_ : Carrier → Carrier → Carrier
```

A $\mathtt{Magma}_0$ **is a** pair $\langle \mathtt{C},\ \mathtt{op} \rangle$ of a type `C` and an operation `op` on that type!

A $\mathtt{Magma}_1$ **on** a given type `C` **is a** one-tuple $\langle \mathtt{op} \rangle$ consisting of a binary operation on that type!

$\mathtt{Magma}_0 \cong (\Sigma\ \mathtt{C} : \mathtt{Set} \bullet \mathtt{Magma}_1\ \mathtt{C})$

```
{- 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
```

In **theory**, parameterised structures are no different from their unparameterised, or “bundled”, counterparts. Indeed, we can easily prove $\mathtt{Magma}_0 \cong (\Sigma\ \mathtt{C} : \mathtt{Set} \bullet \mathtt{Magma}_1\ \mathtt{C})$ by “packing away the parameters” and $\forall\ (\mathtt{C} : \mathtt{Set}) \to \mathtt{Magma}_1\ \mathtt{C} \equiv (\Sigma\ \mathtt{M} : \mathtt{Magma}_0 \bullet \mathtt{M.Carrier} \equiv \mathtt{C})$ 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.

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.

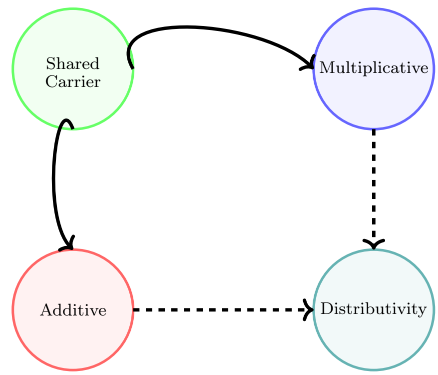
Distributivity is Difficult to Express

```
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 $\mathtt{distribute}_0$.

Even though the carriers of the structures are propositionally equal, $\texttt{R}_{+} \equiv \texttt{R}_{\times}$, they are not the same by definition —the notion of equality was defined in section **??**. 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 $\texttt{distribute}_0$ 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 *informal* diagram to the right —where the dashed lines denote inclusions or dependency relationships.

Bundled forms require (curved) coercisions



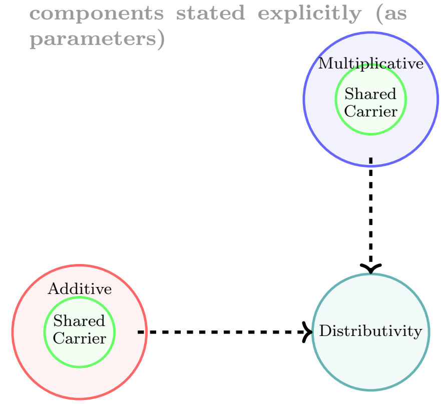
Again, in theory, parameterised structures are no different from their unparameterised, or “bundled”, counterparts. However, in **practice**, even when multiple presentations of an idea are *equivalent* in some sense, there may be specfic presentations that are *useful* for particular purposes[8] . That is, in a depeendely-typed language, equivalence of structures and their usability profiles do not necessarily 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.

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

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)
```

Unbundled forms have shared components stated explicitly (as parameters)



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.

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, ??, ??.

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

The distributivity axiom, from above, 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.

For example, 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.

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:

1. `Monoid`
2. `Group`
3. `AbelianGroup`
4. `CommutativeMonoid`
5. `SemigroupWithoutOne`
6. `NearSemiring`
7. `Semiring`
8. `CommutativeSemiringWithoutOne`
9. `CommutativeSemiring`
10. `CommutativeRing`

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

```
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

```
record Semigroup c ℓ : Set (suc (c ⊔ ℓ)) where
  infixl 7 _·_
  infix  4 _≈_
  field
    Carrier     : Set c
    _≈_         : Rel Carrier ℓ
    _·_         : Op₂ Carrier
    isSemigroup : IsSemigroup _≈_ _·_
```

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 $\Sigma\ \mathtt{X} : \mathcal{X} \bullet \mathcal{X}.\mathtt{f}\ \mathtt{X} \equiv \mathbf{f}_0$ to *specialise* the feature $\mathcal{X}.\mathtt{f}$ to the particular choice $\mathbf{f}_0$. However, nearly all uses of this type will be of the form `(X , refl)` where the `refl` proof is unhelpful noise.

Since the standard library uses the predicate pattern, `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 `Is`$\mathcal{X}$ but only[9] takes i-many elements —c.f., $\mathtt{Magma}_0$ and $\mathtt{Magma}_1$ 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 [13], we shall show how the bundled form $\mathcal{X}$ acts as ***the*** definition, with other forms being derived-as-needed.

[9] 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`.

[13] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. "A language feature to unbundle data at will (short paper)". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*. Ed. by Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm. ACM, 2019, pp. 14–19. ISBN: 978-1-4503-6980-0. DOI: 10.1145/3357765.3359523. URL: https://doi.org/10.1145/3357765.3359523

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 non-trivial due to the need to rewrite along propositional equalities, as was the case with simply stating the distributivity law using $\mathtt{Magma}_0$. Another problem with fully bundled structures is that accessing deeply nested components requires lengthy 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[α] are flexible in theory, but in practice one must enumerate all components to actually state and apply results about such structures.

[α] As in $\mathcal{X}_n$, for n the number of sort and function symbols of the structure.

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 **??**, 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 [20] discusses the numerous problems of bundled presentations as well as the issues of unbundled presentations and settles on using typeclasses along with their tremendously 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, declaratively,* 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 **??**.

[20] Bas Spitters and Eelis van der Weegen. "Type classes for mathematics in type theory". In: *Mathematical Structures in Computer Science* 21.4 (2011), pp. 795–825. DOI: 10.1017/S0960129511000119. URL: https://doi.org/10.1017/S0960129511000119

Notice that to phrase the distributivity 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, '$\sqcup$', to denote idempotent operations.

$0 \times a \approx a$ The identity of "multiplicative symbols" should never resemble '0'; instead it should resemble '1' or, at least, 'e'.

The use of e is a standard, abbreviating *einheit* which means *identity*, as used in influential algebraic works of German authors.

$f + g$ The *sequential* composition of functions is almost universally denoted by multiplicative symbols, such as '$\circ$', '$⨾$', and '$\cdot$'.

Even if monoids are defined with the prototypical binary operation denoted '+', it would be *'wrong'* to continue using it to denote functional composition.

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 $\times, \cdot, *$ 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" [1],
2. The "RATH-Agda" library [17], and
3. A recent "agda-categories" library [16].

[1] *Agda Standard Library.* 2020. URL: `https://github.com/agda/agda-stdlib` (visited on 03/03/2020)

[17] Wolfram Kahl. *Relation-Algebraic Theories in Agda.* 2018. URL: `http://relmics.mcmaster.ca/RATH-Agda/` (visited on 10/12/2018)

[16] Jason Hu Jacque Carrette. *agda-categories library.* 2020. URL: `https://github.com/agda/agda-categories` (visited on 08/20/2020)

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 reappear later, section 4.3, as a library method.

The "Shape" of Renaming Blocks in Agda

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

The content itself is not important itself: The focus is on the renaming that takes place. As such, going forward, we intentionally render such clauses in a tiny fontsize.

Keep an eye out for all those `renaming` (η_1 `to` ${\eta_1}'$; …; η_k `to` ${\eta_k}'$) lines!

3.2.1 Renaming Problems from Agda's Standard Library

Below 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.

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 3^{rd} 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 4^{th} 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.

Seotoid$\mathcal{D}$ Renamings —$\mathcal{D}$ecorated 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˘≡˘⟩_
    )
```

RATH: *For contexts where calculation in different setoids is necessary, we provide "decorated" versions of the* ***`Setoid′`*** *and* ***`SetoidCalc`*** *interfaces [. . .]*

This keeps going to cover the entirety of the English alphabet `SetoidD`, `SetoidE`, `SetoidF`, . . . , `SetoidZ` then we shift to a *few* subscripted versions `Setoid`$_0$, `Setoid`$_1$, . . . , `Setoid`$_4$.

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

SeotoidCalc$\mathcal{D}$ Renamings —$\mathcal{D}$decorated 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_
    )
```

This keeps going to cover the entire English alphabet `SetoidCalcC`, `SetoidCalcD`, `SetoidCalcE`, . . . , `SetoidCalcZ` then we shift to subscripted versions `SetoidCalc`$_0$, `SetoidCalc`$_1$, . . . , `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`$\mathcal{XXX}$ takes around 10 lines, for a total of roughly 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`$\mathcal{D}$ and `LocalSetoidCalc`$\mathcal{D}$ where decoration $\mathcal{D}$ ranges over $_0$, $_1$, $_2$, $_3$, $_4$, `R`. The inconsistency in not providing the other decorations used for `Setoid`$\mathcal{D}$ 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[α] ignored the need for meaningful names altogether. Below are a few notable instances.

α Perhaps naming was ignored for the sake of quick development and new names may be used in a later relsease.

- ⋄ Functors have fields named `F`$_0$`,` `F`$_1$`,` `F-resp-≈,` ....
  - ∘ This could be considered reasonable even if one has a functor named `G`.

More meaningful names may be `obj, mor, mor-cong` —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 peculiar projection name 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.

   If we declare a few names, the type reads: If $\beta \approx_0 \gamma$ then `η β B ⟨$⟩ f ∘ id` $\approx_1$ `F₁ F f ⟨$⟩ (η γ A ⟨$⟩ id)`. This is just a naturality condition, which are ubiquitous in categroy theory.

   Declare $\_\approx_0\_$ and $\_\approx_1\_$ to be `Setoid._≈_ (F₀ Nat[Hom[C][-,c],F] (F , A))` and, respectively, `Setoid._≈_ (F₀ F B)`.

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 η_1, η_2 or η_s, η_t.

   The subscripts are for 's'ource/1 and 't'arget/2, for a morphism

   $$f : \mathsf{source}\ f \to \mathsf{target}\ f$$

   or $f : X_1 \to X_2$.

   The sequence of f's " `F₁ F f` " looks strange at a first glance; with the alternative suggested naming it just denotes `mor F f`.

   ⋆ ⋆ ⋆

   Just an application of a functor's morphism mapping.

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$^{\alpha}$ and so we need to drop them to obtain the weaker notion of `Semigroup`. 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* [19]. 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. Tersely put: One axiomatisation may be ideal for verifying instances, whereas an equivalent but possibly longer axiomatisation may be more amicable for calculation and computation.

α 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.

[19] Robert C. Martin. *Design Principles and Design Patterns.* Ed. by Deepak Kapur. 1992. URL: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf (visited on 10/19/2018)

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`. 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, to the right we have presented 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'.

In particular, the Category type and the natural isomorphism type are instances of such a pattern.

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

smart : (C : Set) (_⨾_ : C → C → C)
       (assocʳ : ∀ {x y z}
               →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z))
       →
         Semigroup
smart C _⨾_ assocʳ = 𝒮 C _⨾_ assocʳ (sym assocʳ)

-- The opposite of the opposite
-- is definitionally equal to the original

_˘ : 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[β] are bound to arise. Such issues could be reduced, if not avoided, if library methods could have been used instead of manually implementing design patterns.

[β] 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.

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; e.g., quoting RATH-Agda (section 15.13), *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.* For instance, the above semigroup type could have been augmented with an ordering if we view _⨾_ as a meet-operation. Instead, we could 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 }
```

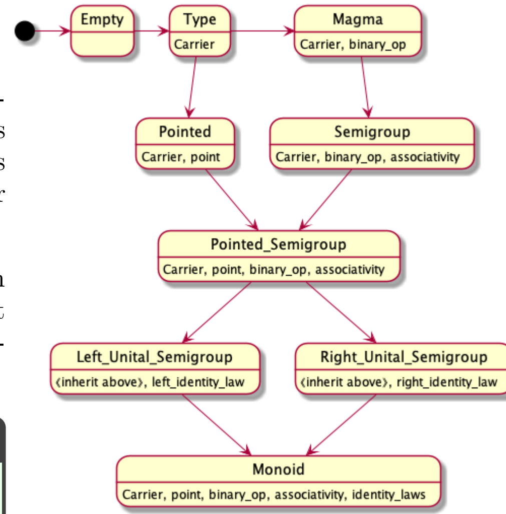
Efficient Redundancy Design Pattern (RATH-Agda section 17.1)

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 of 'ripping things out', in an ideal world, it may be 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*.



Possible hierarchy leading to Monoid

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
```

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
```

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` to the right, needs to define the binary operation in a `Semigroup` instance first, which lives in a `PointedSemigroup` instance, which lives in a `LeftUnitalSemigroup` instance.

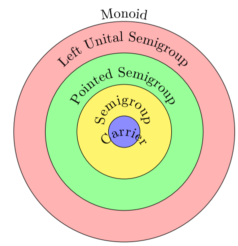
This nesting scenario happens rather often, in one guise or another. The amount of syntactic noise required to produce a simple instantiation is unreasonable: *One should not be forced to work through the hierarchy if it provides no immediate benefit.*

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

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 require deep —'staircase'— projections

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



Extension Design Pattern

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

Extension Design Pattern Prototype

```
record 𝒴 : Set₁ where
  field x : 𝒳
  open  𝒳 x public -- (⋆)
  field f₀ : ⋯
    ⋮
  field fₙ : ⋯
```

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". That is, a 'flattened' hierarchy may be *seen* as 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.

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.

3.5 Conclusion

After 'library spelunking', we are now in a position to summarise the problems encountered, when using existing[10] 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.

[10] 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. Let's consider two concrete examples.

Example (1). 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.

That sounds like a terrific idea! We do it in the next chapter ;-)

Example (2). Perhaps more common is the renaming of package contents, by hand. For example, when a notion of preorder is defined with a relation named `_≤_`, one may rename it and all references to it by, say, `_⊑_`. Again, a pre-processor or editor-tactic could be utilised; yet many simply perform the re-write by hand.

"By hand" 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 discussed in this chapter, using monoids as the prototypical structure. Some patterns we did not cover, as they will be covered in future sections.

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

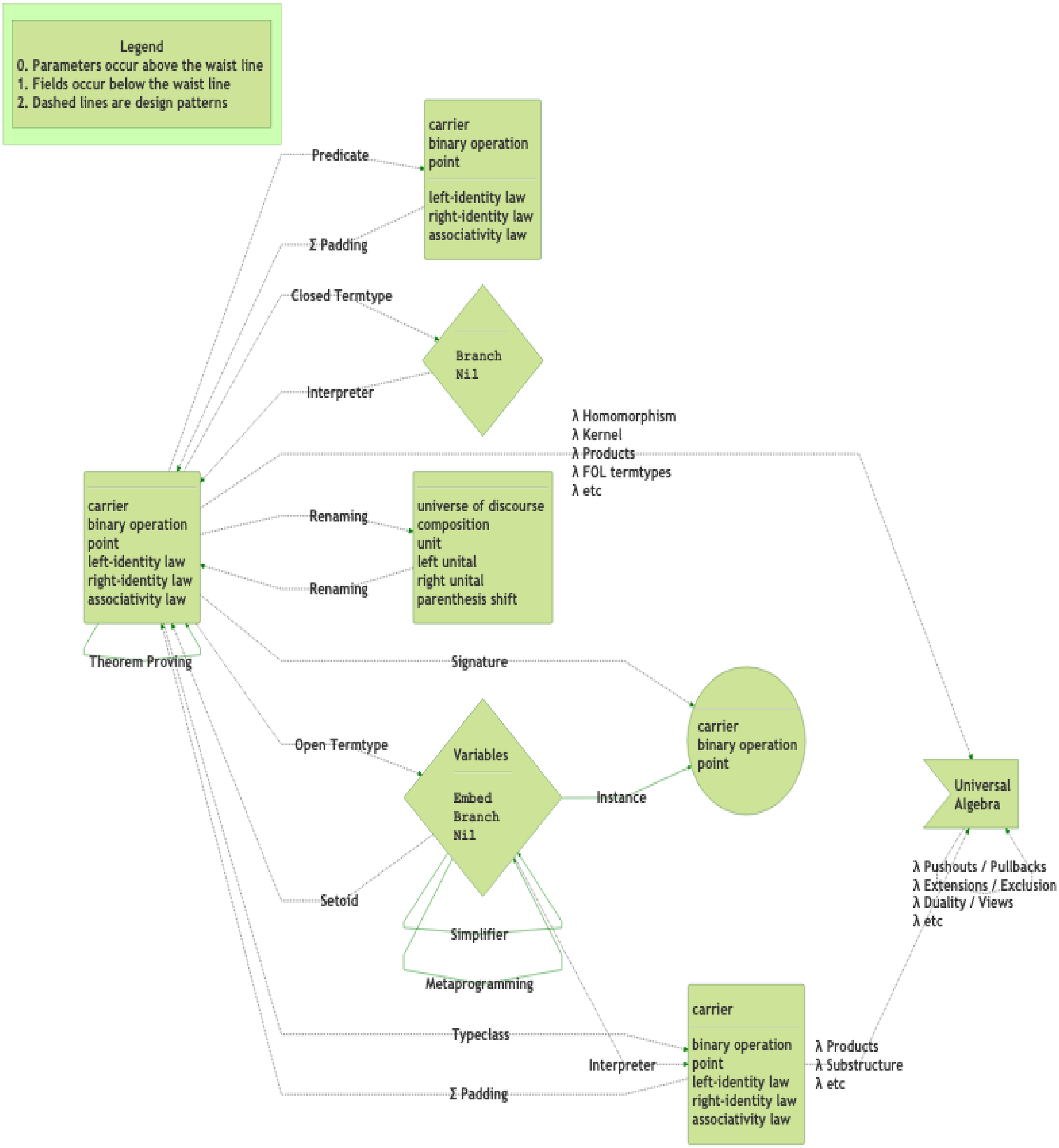


Figure 3.1: PL Research is about getting free stuff: From the left-most node, we can get a lot!

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 a reasonable 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 a small number of 'meta-primitives'[0] 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 not only to show that the 'flattening' of value terms and module terms is feasible[1] ; but to also show that ubiquitous packaging combinators can be generated[2] from a small number of primitives. 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.

[0]Section 4.3 contains an example-driven approach

[1]Indeed, the MathScheme [4] prototype already shows this.

[2]Just as the primitive of a programming language permit arbitrarily complex programs to be written.

Chapter Contents

The core of this chapter shows how some of the problems of Chapter 3, *Examples from the wild*, can be solved using `PackageFormer`.

4.1 Why an editor extension?

The prototype[3] *rewrites* Agda phrases from an extended Agda syntax to legitimate existing syntax; it is written as an Emacs editor extension to Emacs' Agda interface, using Lisp [10]. Since Agda code is predominately written in Emacs, a practical and pragmatic editor extension would need to be in Agda's de-facto IDE[4] , Emacs. Moreover, Agda development involves the manipulation of Agda source code by Emacs Lisp —for example, for case splitting and term refinement tactics— and so it is natural to extend these ideas. Nonetheless, at a first glance, it is humorous[5] 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.*

Unless a language provides an extension mechanism, one is forced to either alter the language's compiler or to use a preprocessing tool —both have drawbacks. The former[6] 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*[7] : It can be a nuisance to remember always invoke a preprocessor before compilation or typechecking, 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 always possible.

However, if the language's community subscribes to *one* 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.

[3] A prototype's raison d'etre is a testing ground for ideas, so its ease of development may well be more important than its usability.

[10] Paul Graham. *ANSI Common Lisp*. USA: Prentice Hall Press, 1995. ISBN: 0133708756

Why Emacs?

[4] **IDE**: Interactive Development Environment

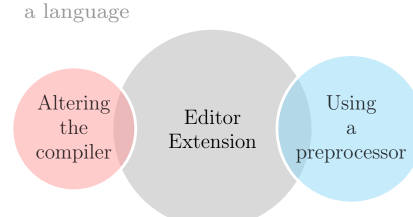
[5] None of my colleagues thought Lisp was at all the 'right' choice; of-course, none of them had the privilege to use the language enough to appreciate it for the wonder that it is.

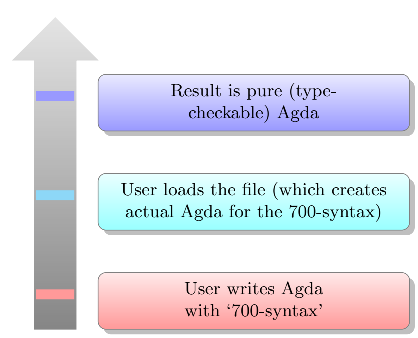
Why an editor extension? Because we quickly needed a *convenient* prototype to actually "figure out the problem".

[6] Instead of "hacking in" a new feature, one could instead carefully research, design, and implement a new feature.

[7] Unless one uses a sufficiently flexible IDE that allows the seemless integration of preprocessing tools; which is exactly what we have done with Emacs.

A reasonable middle path to growing a language





How does it work? All stages transpire in *one* user-written file

Why Lisp? Emacs is extensible using Elisp[8] 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[9] , to quickly see what our extended Agda syntax transpiles into.

[8]Emacs Lisp is a combination of a large porition of Common Lisp and a editor language supporting, e.g., buffers, text elements, windows, fonts.

[9]E.g., since Emacs is 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`.

Finally, Lisp uses a rather small number of constructs, such as macros and lambda, which themselves are used to build 'primitives', such as `defun` for defining top-level functions [15]. Knowing this about Lisp encourages us to emulate this expressive parsimony.

[15] Doug Hoyte. *Let Over Lambda.* Lulu.com, 2008. ISBN: 1435712757

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.

With the extension, Agda's usual `C-c C-l` command parses special comments containing fictitious Agda declarations, produces an auxiliary Agda file which it ensures is imported in the current file, then control is passed to the usual Agda typechecking mechanism.

Below is example code that can occur in the specially recognised comments. 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 **??**, 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'.

In the code block, 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**.

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
9                                           ≡   a · (b · v)
```

Different Ways to Organise ("interpret" / "use") M-Sets

```
Semantics   = M-Set ⟴ record
Semantics𝒟 = Semantics ⟴ rename (λ x → (concat x "𝒟"))
Semantics₃ =  Semantics :waist 3

Left-M-Set   = M-Set ⟴ record
Right-M-Set = Left-M-Set ⟴ flipping "_·_" :renaming "leftId
↪  to rightId"

ScalarSyntax = M-Set ⟴ primed ⟴ data "Scalar′"
Signature     = M-Set ⟴ record ⟴ signature
Sorts         = M-Set ⟴ record ⟴ sorts

𝒱-one-carrier    = renaming "Scalar to Carrier; Vector to
↪  Carrier"
𝒱-compositional = renaming "_×_ to _⨾_; _·_ to _⨾_"
𝒱-monoidal       = one-carrier ⟴ compositional ⟴ record

LeftUnitalSemigroup = M-Set ⟴ monoidal
Semigroup           = M-Set ⟴ keeping "assoc" ⟴ monoidal
Magma               = M-Set ⟴ keeping "_×_" ⟴ monoidal
```

Now to actually use this context …

M-Sets as records, possibly with renaming or parameters.

⋆ ⋆ ⋆

Duality; we might want to change the order of the action, say, to write `evalAt x f` instead of `run f x` —using the program-input interpretation of M-Sets above.

⋆ ⋆ ⋆

Keeping only the 'syntactic interface', say, for serialisation or automation.

⋆ ⋆ ⋆

Collapsing different features to obtain the notion of "monoid".

⋆ ⋆ ⋆

Obtaining parts of the monoid hierarchy (see chapter 3) from M-Sets

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.

Let's discuss what's actually going on here.

The first line declares the context of `M-Set`s using traditional Agda syntax " `record M-Set : Set₁ where` " except the we use the word `PackageFormer` to avoid confusion with the existing record concept, but[10] we also *omit* the need for a `field` keyword and *forbid* the existence of parameters. Such abstract contexts have no concrete form in Agda and so no code is generated; the second snippet above[11] shows sample declarations that result in legitimate Agda.

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

[10] **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.

[11] For every (special 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.

In line 9, 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`.

The waist is the number of parameters exposed; recall $\Pi^w\Sigma$ from chapter 2.

Elaboration of lines 9-11 — Record / decorated renaming / typeclass forms

```
{- Semantics          = M-Set ⟴ record -}
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)
```

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[12] —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` → `Left-M-Set`.

[12]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.

"Arbitrary functions act on modules": When only one variational is applied to a context, the one and only sequencing operator '⟴' may be omitted. As such, the $\mathcal{D}$ecorated `Semantics𝒟` 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.

Likewise, line 13, mentions another combinator

```
_flipping_ : PackageFormer → Name → PackageFormer
```

All combinators are demonstrated in this section and their usefulness is dicussed in the nextion section. For example, in contrast to the above 'type', the `flipping` combinator also takes an *optional keyword argument* `:renaming`, which simply renames the given pair. The notation of keyword arguments is inherited from Lisp.

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.

Elaboration of lines 13-14 Duality: Sets can act on semigroups from the left or the right

```
{- Left-M-Set          = M-Set ⟴ record -}
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}
```

Next, in line 16, we view a context as such a termtype by declaring one sort of the context to act as the termtype (carrier) 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.

An algebraic data type is a tagged union of symbols, terms, and so is one type —see section ??.

Recall from Chapter ??, 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**.

Elaboration of lines 16-18 — Termtypes and lawless presentations

```
{- ScalarSyntax        = M-Set ⟴ primed ⟴ data "Scalar′" -}
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 in `ScalarSyntax` is needed so that the names 𝟙, `_×_` do not pollute the global name space.

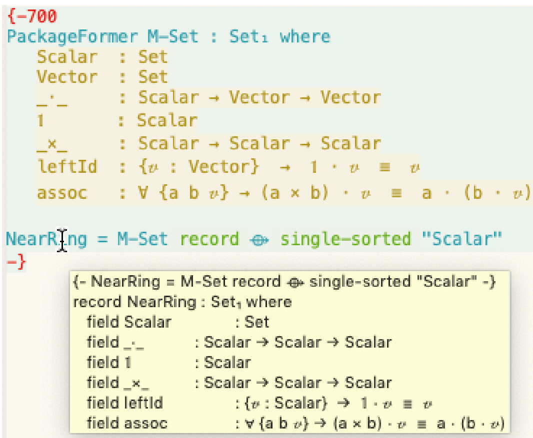
Finally, starting with line 20, declarations start with “ 𝒱- ” to indicate that a new variation *combinator* is to be formed, rather than a new *grouping* mechanism. For instance, the user-defined `one-carrier` variational identifies both the `Scalar` and `Vector` sorts, whereas `compositional` identifies the binary operations; then, finally, `monoidal` performs both of those operations and also produces a concrete Agda `record` formulation. Below, in the final code snippet of this section, are the elaborations of using these new new user-defined variationals.

User defined variationals are applied as if they were built-ins.

Elaboration of lines 24-26 — Conflating features gives familiar structures

```
{- LeftUnitalSemigroup = M-Set ⟴ monoidal -}
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 the figure below, the source file is furnished with tooltips displaying the special 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 special 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.



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

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 [3], reiterating many the ideas therein. These features are **not built-in** but instead are constructed from a small set of primitives, shown below, just as a small core set of language features give way to complex software programs. Moreover, users may combine the primitives —using Lisp— to **extend** the system to produce grouping mechanisms for any desired purpose.

[3] Jacques Carette and Russell O'Connor. "Theory Presentation Combinators". In: *Intelligent Computer Mathematics* (2012), pp. 202–215. DOI: 10.1007/978-3-642-31374-5_14

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.

The few constructs demonstrated in this section not only create new grouping mechanisms from old ones, but also create morphsisms from the new, child, presentations to the old parent presentations. For example, a theory extended by new declarations comes equipped with a 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 primitives —but we have implemented a few to show that the primitives are deserving of their name, as shown below.

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` $\to$ `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

`PackageFormer` packages are an **implementation of the idea** of packages fleshed out in Chapter **??**. 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 τ can be thought of as a **binary packaged-valued operator**,

```
_v_ : PackageFormer
    → τ
    → PackageFormer
```

With this perspective, the *sequencing variational combinator* '$\oplus\!\!\!\rangle$' 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 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 **??**— 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[13] 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.

[13] 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 the basics is provided.

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 `𝟙-unique`— and `CommutativeMonoid₀` below.

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 → 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.

Manually Repeating the entirety of 'Monoid' within 'CommutativeMonoid₀'

```
PackageFormer Monoid : Set₁ where
   Carrier : Set
   _·_     : Carrier → Carrier → Carrier
   assoc   : {x y z : Carrier} → (x · y) · z  ≡  x · (y · z)
   𝟙       : Carrier
   leftId  : {x : Carrier} → 𝟙 · x  ≡ x
   rightId : {x : Carrier} → x · 𝟙  ≡ x
   𝟙-unique : ∀ {e} (lid : ∀ {x} → e · x ≡ x) (rid : ∀ {x} →
   ↪  x · e ≡ x) → e ≡ 𝟙
   𝟙-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)
   𝟙       : Carrier
   leftId  : {x : Carrier} →  𝟙 · x  ≡ x
   rightId : {x : Carrier} →  x · 𝟙  ≡ x
   comm    : {x y : Carrier} →  x · y  ≡  y · x
   𝟙-unique : ∀ {e} (lid : ∀ {x} → e · x ≡ x) (rid : ∀ {x} →
   ↪  x · e ≡ x) → e ≡ 𝟙
   𝟙-unique lid rid = ≡.trans (≡.sym leftId) rid
```

So much repetition for an additional axiom! Eek!

As expected, the only difference is that $\texttt{CommutativeMonoid}_0$ adds a commutatity axiom. Thus, given Monoid, it would be **more economical** to define:

Economically declaring only the new additions to 'Monoid'

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

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**.

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 $\texttt{CommutativeMonoid}_0$ above —followed by forgetful operation. The tooltip shows the *expanded* version of the theory, which is **what we want to specify but not what we want to enter manually**.

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 then becomes an issue. Instead, as shown below, 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.

⋆ ⋆ ⋆

Method call (s-replace old new s) replaces all occurrences of string old by new in the given string s.

⋆ ⋆ ⋆

(pcase e (x_0 y_0) ... (x_n y_n)) pattern matches on e and performs the first y_i if e = x_i, otherwise it returns nil.

Writing definitions **only once** with 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."
   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")
      ("congruence"    "cong    : ∀ x x′ y y′ → rel x x′ → rel y y′ → rel (op x x′) (op y y′)")
      (_ (error "𝒱-postulating does not know the property ‘%s’" prop))
      )))) :adjoin-retract 'adjoin-retract)
```

As such, we have a formal approach to the idea that **each piece of mathematical knowledge should be formalised only once** [9]. We can extend this database of properties as needed with relative ease. Here is an example use along with its elaboration.

[9] Adam Grabowski and Christoph Schwarzweller. "On Duplication in Mathematical Repositories". In: *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings.* Ed. by Serge Autexier et al. Vol. 6167. Lecture Notes in Computer Science. Springer, 2010, pp. 300–314. ISBN: 978-3-642-14127-0. DOI: 10.1007/978-3-642-14128-7_26. URL: https://doi.org/10.1007/978-3-642-14128-7%5C_26

Example Use

```
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` ⋯ 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`.

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 "𝟙"
Monoid              = LeftUnitalSemigroup postulating "_·_" "right-unit" :using "𝟙"
CommutativeMonoid   = Monoid              postulating "_·_" "commutative"
```

Of-course, one can continue to build packages in a monolithic fashion, as shown below.

```
Group = Monoid extended-by "_⁻¹ : Carrier → Carrier; left⁻¹ : ∀ {x} → (x ⁻¹) · x ≡ 𝟙;
↪  right⁻¹ : ∀ {x} → x · (x ⁻¹) ≡ 𝟙" ⟴ record
```

After discussing renaming, we return to discuss the loss of relationships when we augment `Group` with a commutativity axiom —commutative groups are commutative monoids!

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

```
AbealianMonoid = CommutativeMonoid 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 `toCommutativeMonoid`, and still has the *inherited* projection `toMonoid`.

There are a few reasonable properties that a renaming construction should support. Let us briefly look at the (operational) 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.

Adjoining coretracts —views from parent packages

```
Sequential = Magma renaming "op to _⨾_" :adjoin-coretract t
```

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.

Sequential 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}
```

This user implementation of `renaming` avoid name clashes for λ-arguments by using *gensyms* —generated symbolic names, "fresh variable names".

Commutativity. Since `renaming` and `postulating` both adjoin retract morphisms, by default, we are led 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[14] .

[14] For instance, we may define idempotent magmas with

```
renaming "_·_ to _⊔_"
⟴ postulating "_⊔_" "idempotent"
:adjoin-retract nil
```

or, equivalently (up to reordering of constituents), with

```
postulating "_⊔_" "idempotent"
⟴ renaming "_·_ to _⊔_"
:adjoin-retract nil
```

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

```
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 _⊔_"
```

Finally, as expected, simultaneous renaming works too, and renaming is an invertible operation —e.g., below `Magma`^{rr} is identical to `Magma`.

(Recall `renaming′` performs renaming but does not adjoin retract views.)

```
Magmaʳ  = Magma  renaming′ "_·_  to op"
Magmaʳʳ = Magmaʳ renaming′ "op   to _·_"
```

`TwoR` is just `Two` but as an Agda `record`, so it typechecks.

Simultaneous textual substitution example

```
PackageFormer Two : Set₁ where
  Carrier : Set
  𝟘       : Carrier
  𝟙       : Carrier

TwoR = Two record ⟴ renaming′ "𝟘 to 𝟙; 𝟙 to 𝟘"
```

Do-it-yourself. Finally, to demonstrate the accessibility of the system, we show how a generic renaming operation can be defined swiftly using the primitives mentioned listed in the first table of this section. 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)
```

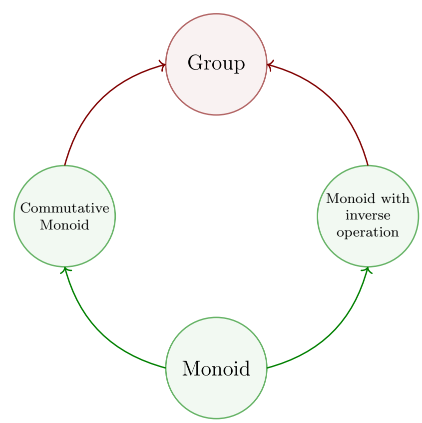
4.3.4 Unions/Pushouts (and intersections)

But even with these features, using `Group` from above, we would find ourselves writing:

```
CommutativeGroup₀ = Group 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; `CommutativeGroup₀` then closes this diamond-loop by adding both features, as shown in the figure to the right. The simplest way to share structure is to union two presentations:

Given green, require red



Unions of packages

```
CommutativeGroup = Group union CommutativeMonoid ⟴ record
```

The resulting record, `CommutativeMonoidR`, comes with three[15] 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 [3]. 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

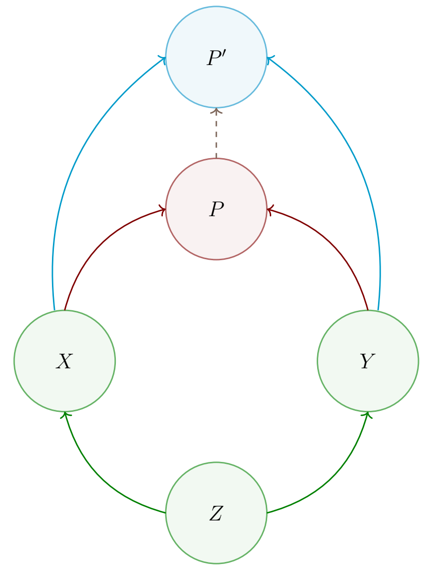
[15] The three green arrows in the diagram above!

[3] Jacques Carette and Russell O'Connor. "Theory Presentation Combinators". In: *Intelligent Computer Mathematics* (2012), pp. 202–215. DOI: 10.1007/978-3-642-31374-5_14

as optional keyword arguments. The more general notion of pushout is required if we were to combine[16] `Group` with `AbealianMonoid`, which have non-identical syntactic copies of `Monoid`.

[16]For example, to make rings!

What is a pushout?



Given green, require red, such that every candidate cyan has a unique umber

The pushout of morphisms $f : Z \to X$ and $g : Z \to Y$ is, essentially, the disjoint sum of contexts 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 the figure to the right, 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. Hence, 6 inputs are actually needed for forming a *usable* pushout object.

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 $\mathsf{inj}_1 : X \to P$, $\mathsf{inj}_2 : Y \to P$ 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 $\mathsf{toX} : P \to X$ and $\mathsf{toY} : P \to Y$ 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!*

By changing perspective, we half the number of inputs to the pushout construction!

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 input 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* $f'\,x = z = g'\,y$.

Incidentally, using the reversed 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 them *optional* arguments.

Before we show the implementation of `union`, let us showcase an example that mentions all arguments, optional and otherwise —i.e., test-driven development. Besides the elaboration The **commutative** diagram, to the right, *informally* carries out the `union` construction that results in the elaborated code below.

Bimagmas: Two magmas sharing the same carrier

```
BiMagma = Magma union Magma :renaming₁ "op to _+_" :renaming₂
↪  "op to _×_"  :adjoin-retract₁ "left" :adjoin-retract₂
↪  "right"
```

Elaboration

```
record BiMagma : 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 = _×_}
```

That is, *this particular user implementation* realises

`X₁ union X₂ :renaming₁ f′ :renaming₂ g′`

as the pushout of the inclusions

$$\texttt{f}'\ \texttt{X}_1 \cap \texttt{g}'\ \texttt{X}_2 \hookrightarrow \texttt{X}_i$$

where the source is the set-wise intersection of *names*. Moreover, when either `renaming`$_i$ is omitted, it defaults to the identity function.

In Lisp, optional keyword arguments are passed with the syntax `:arg val`.

⋆ ⋆ ⋆

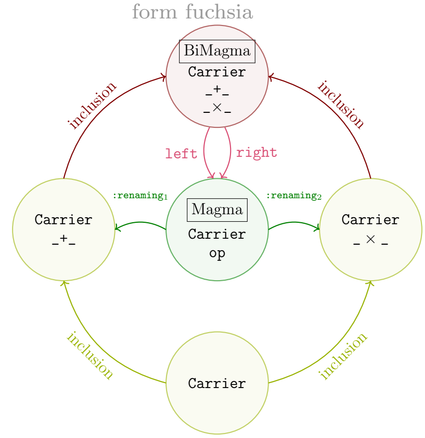
Invoke `union` with `:adjoin-retract`$_i$ `"new-function-name"` to use a new name, or `nil` instead of a string to omit the retract —as was done for `extended-by` earlier.

⋆ ⋆ ⋆

Whew, a worked-out example!

The user manual contains full details and an implementation of intersection, pullback, as well.

Given green, yield yellow, require red, form fuchsia



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}
```

Idempotence. The main reason that the construction is named ‘union’ instead of ‘pushout’ is that, modulo adjoined retracts, it is idempotent. For example, `Magma union Magma` ≈ `Magma` —this is essentially the previous bi-magma example *but* we are not distinguishing (via `:renaming`$_i$) the two instances of `Magma`.

Disjointness. On the other extreme, distinguishing all the names of one of the input objects, we have disjoint sums. In contrast to the above bi-magma, in the example below, we are not distinguishing the two instances of `Magma` 'on the fly' via `:renaming`$_i$ but instead making them disjoint beforehand using `primed` —which is specified informally as `p primed ≈ p :renaming (λ name → name ++ "′")`.

```
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′}
```

Before returning to the diamond problem, we show an implementation not so that the reader can see some cleverness —not that we even expect the reader to understand it— but instead to showcase that a sufficiently complicated combinator, which is *not built-in*, can be defined without much difficulty.

(Abridged) Pushout combinator with 4 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)))

       (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)))))))
```

Indeed, the core of the construction lies in the first 12 lines of the `let*` clause; the rest are extra bells-and-whistles —which could have been omitted, by the user, for a faster implementation.

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.

Support for Diamond Hierarchies

A common scenario is extending a structure, say `Magma`, into orthogonal directions, such as by making its 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

```
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.

```
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)
```

This example, as well as mitigating diamond problems, show that the implementation outlined is reasonably well-behaved.

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. Classically in Agda, duality is *utilised* as follows:

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`.

1. Define a *parameterised* module `R _·_` for the desired ideas **on** the operation `_·_`.

Example

```
module R (_·_ : X → Y → Z) where
    ·-isLeftId : X → Set
    ·-isLeftId e = ∀ {x} → e · x ≡ x
```

2. Define a shallow (parameterised) module `R`^{op} `_·_` that essentially only opens `R _·`^{op}`_` and renames the concepts in `R` with dual names.

Continuing...

```
module Rᵒᵖ (_·_ : X → Y → Z) where
    public open R _·_
           renaming (·-isLeftId to ·-isRightId)
```

The RATH-Agda [17] 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.

The ubiquity of duality!

[17] Wolfram Kahl. *Relation-Algebraic Theories in Agda.* 2018. URL: http://relmics.mcmaster.ca/RATH-Agda/ (visited on 10/12/2018)

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. 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`.

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.

Consider the following heterogeneous algebra —which is essentially the main example of section 4.2 but missing the associativity field.

Left unital actions

```
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

```
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 to*Parent* 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

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* [6] 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* [19, 7] : *No client should be forced to depend on methods it does not use.*

Tiny Theories Example

```
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"
```

[6] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. "Little theories". In: *Automated Deduction—CADE-11*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 567–581. ISBN: 978-3-540-47252-0

[19] Robert C. Martin. *Design Principles and Design Patterns*. Ed. by Deepak Kapur. 1992. URL: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf (visited on 10/19/2018)

[7] Eric Freeman and Elisabeth Robson. *Head first design patterns - your brain on design patterns*. O'Reilly, 2014. ISBN: 978-0-596-00712-6. URL: http://www.oreilly.de/catalog/hfdesignpat/index.html

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

```
Empty′        = Monoid keeping ""
Type′         = Monoid keeping "Carrier"
Magma′        = Monoid keeping "_·_"
Semigroup′    = Monoid keeping "assoc"
PointedMagma′ = Monoid keeping "𝟙; _·_"
                -- This is just “ keeping: Carrier; _·_; 𝟙 ”
```

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

```
LeftUnitalMagma = Monoid keeping "𝟙-unique" ⟴ record
```

This expands to the following theory, minimal enough to derive `𝟙-unique`.

Elaboration

```
record LeftUnitalMagma : Set₁ where

   field
     Carrier : Set
     _·_     : Carrier → Carrier → Carrier
     𝟙       : Carrier
     leftId  : {x : Carrier} → 𝟙 · x  ≡ x

   𝟙-unique     : ∀ {e} (lid : ∀ {x} → e · x ≡ x) (rid : ∀ {x} → x · e ≡ x) → e ≡ 𝟙
   𝟙-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

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, which in turn was inspired

⥁ People should enter terse, readable, specifications that expand into useful, typecheckable, code that may be dauntingly larger in textual size. ⥁

by web lists of Peter Jipsen, John Halleck, and many others from Wikipedia and nLab [3, 4] . 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.

[3] Jacques Carette and Russell O'Connor. "Theory Presentation Combinators". In: *Intelligent Computer Mathematics* (2012), pp. 202–215. DOI: 10.1007/978-3-642-31374-5_14

[4] Jacques Carette et al. *The MathScheme Library: Some Preliminary Experiments.* 2011. arXiv: 1106.1862v1 [cs.MS]

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.

Unlike other systems, PackageFormer does not come with a static set of module operators —it grows dynamically, possibly by you, the user.

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. Nonetheless, we are grateful for the source file provided by the MathScheme team.

MathScheme's design hierarchy raised certain semantic concerns 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.

4.4 Contributions: From Theory to Practice

The `PackageFormer` implements the ideas of Chapters **??** 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, Idris [2] , and even Haskell [18]. The `PackageFormer` implementation has the following useful properties.

[2] Edwin Brady. *Type-driven Development With Idris.* Manning, 2016. ISBN: 9781617293023. URL: http://www.worldcat.org/isbn/9781617293023

[18] Sam Lindley and Conor McBride. "Hasochism: the pleasure and pain of dependently typed haskell programming". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013.* Ed. by Chung-chieh Shan. ACM, 2013, pp. 81–92. ISBN: 978-1-4503-2383-3. DOI: 10.1145/2503778.2503786. URL: https://doi.org/10.1145/2503778.2503786

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`.

   ◇ 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.

Generated modules are necessarily 'flattened' for typechecking with Agda —see section 4.3.1.

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.

Moreover, all of this happens in the *background* preceeding the ussual typechecking command, `C-c C-l`.

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 .

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

—Not yet re-worked into this new marginful format—

6 Conclusion

—Not yet re-worked into this new marginful format—

Bibliography

Here are the references in citation order.

[1] *Agda Standard Library*. 2020. URL: https://github.com/agda/agda-stdlib (visited on 03/03/2020).

[2] Edwin Brady. *Type-driven Development With Idris*. Manning, 2016. ISBN: 9781617293023. URL: http://www.worldcat.org/isbn/9781617293023.

[3] Jacques Carette and Russell O'Connor. "Theory Presentation Combinators". In: *Intelligent Computer Mathematics* (2012), pp. 202–215. DOI: 10.1007/978-3-642-31374-5_14.

[4] Jacques Carette et al. *The MathScheme Library: Some Preliminary Experiments*. 2011. arXiv: 1106.1862v1 [cs.MS].

[5] Edsger W. Dijkstra. *The notational conventions I adopted, and why*. circulated privately. July 2000. URL: http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF.

[6] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. "Little theories". In: *Automated Deduction—CADE-11*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 567–581. ISBN: 978-3-540-47252-0.

[7] Eric Freeman and Elisabeth Robson. *Head first design patterns - your brain on design patterns*. O'Reilly, 2014. ISBN: 978-0-596-00712-6. URL: http://www.oreilly.de/catalog/hfdesignpat/index.html.

[8] François Garillot et al. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. Ed. by Tobias Nipkow and Christian Urban. Vol. 5674. Lecture Notes in Computer Science. Munich, Germany: Springer, 2009. URL: https://hal.inria.fr/inria-00368403.

[9] Adam Grabowski and Christoph Schwarzweller. "On Duplication in Mathematical Repositories". In: *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*. Ed. by Serge Autexier et al. Vol. 6167. Lecture Notes in Computer Science. Springer, 2010, pp. 300–314. ISBN: 978-3-642-14127-0. DOI: 10.1007/978-3-642-14128-7_26. URL: https://doi.org/10.1007/978-3-642-14128-7%5C_26.

[10] Paul Graham. *ANSI Common Lisp*. USA: Prentice Hall Press, 1995. ISBN: 0133708756.

[11] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994. ISBN: 0-201-55802-5. URL: https://www-cs-faculty.stanford.edu/%5C%7Eknuth/gkp.html.

[12] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer, 1993. ISBN: 0-387-94115-0. DOI: 10.1007/978-1-4757-3837-7. URL: https://doi.org/10.1007/978-1-4757-3837-7.

[13] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. "A language feature to unbundle data at will (short paper)". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*. Ed. by Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm. ACM, 2019, pp. 14–19. ISBN: 978-1-4503-6980-0. DOI: 10.1145/3357765.3359523. URL: https://doi.org/10.1145/3357765.3359523.

[14] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books Inc., 1979.

[15] Doug Hoyte. *Let Over Lambda*. Lulu.com, 2008. ISBN: 1435712757.

[16] Jason Hu Jacque Carrette. *agda-categories library*. 2020. URL: https://github.com/agda/agda-categories (visited on 08/20/2020).

[17] Wolfram Kahl. *Relation-Algebraic Theories in Agda*. 2018. URL: http://relmics.mcmaster.ca/RATH-Agda/ (visited on 10/12/2018).

[18] Sam Lindley and Conor McBride. "Hasochism: the pleasure and pain of dependently typed haskell programming". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. Ed. by Chung-chieh Shan. ACM, 2013, pp. 81–92. ISBN: 978-1-4503-2383-3. DOI: 10.1145/2503778.2503786. URL: https://doi.org/10.1145/2503778.2503786.

[19] Robert C. Martin. *Design Principles and Design Patterns*. Ed. by Deepak Kapur. 1992. URL: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf (visited on 10/19/2018).

[20] Bas Spitters and Eelis van der Weegen. "Type classes for mathematics in type theory". In: *Mathematical Structures in Computer Science* 21.4 (2011), pp. 795–825. DOI: 10.1017/S0960129511000119. URL: https://doi.org/10.1017/S0960129511000119.