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

August 30, 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.

Contents

1 Introduction

unchanged

2 Packages and Their Parts

unchanged

Prerequisite of the reader

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

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

3 Examples from the Wild

unchanged

4 The `PackageFormer` Prototype

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

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

Chapter Contents

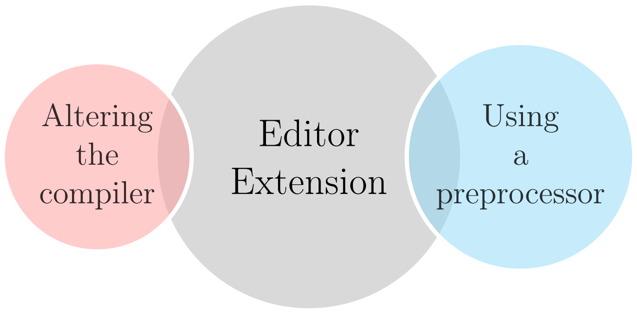
4.1 Why an editor extension? Why Lisp is reasonable?

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

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

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

Figure 4.1: A reasonable middle path to growing a language

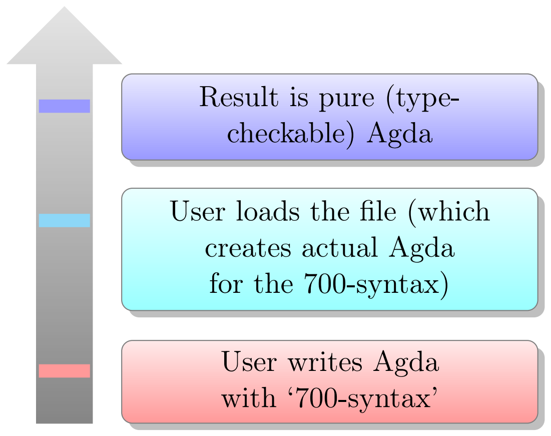


to *plug-in* the necessary preprocessing —to transform the extended language into the pure core language— in a saliently **silent** fashion such that users need not invoke it manually. Moreover, to mitigate the burden of increasing the toolchain, the salient preprocessing would **not transform user code** but instead **produce auxiliary files** containing core language code which are then *imported* by user code —furthermore, such import clauses could be automatically inserted when necessary. The benefit here is that **library users** need not know about the extended language features; since all files are in the core language with extended language feature appearing in special comments. Details can be found in section 4.2, while Figure 4.1 provides a bird's eye view.

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

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

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


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

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

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

which may be written in Haskell as

```
map (λ it → it + 2) [1, 2, 3]
```

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

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

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

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

4.2 Aim: *Scrap the Repetition*

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

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

1. Parse any comments `{-700 ··· -}` containing fictitious Agda code,
2. Produce legitimate Agda code for the '700-comments' into a file `X_generated.agda`,
3. Add to `X.agda` a call to import `X_generated.agda`, if need be; and, finally,
4. Actually perform the expected typechecking.

   ◇ For every 700-comment declaration $\mathcal{L}$ = $\mathcal{R}$ in the source file, the name $\mathcal{L}$ obtains a tooltip which mentions its specification $\mathcal{R}$ and the resulting legitimate Agda code. This feature is indispensable as it lets one generate grouping mechanisms and quickly ensure that they are what one intends them to be.

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

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

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

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

Different Ways to Organise ("interpret"

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

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

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

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

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

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

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

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

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

Conflating fields, parameters, and definitional extensions

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

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

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

Arbitrary functions act on modules

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

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

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

Record / decorated renaming / typeclass forms

```
{- 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)
```

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

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

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

```
{- 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}
```

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

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

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

Termtypes and lawless presentations

```
{- 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 is needed so that the names `𝟙, _×_` do not pollute the global name space.)

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

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

Conflating features gives familiar structures

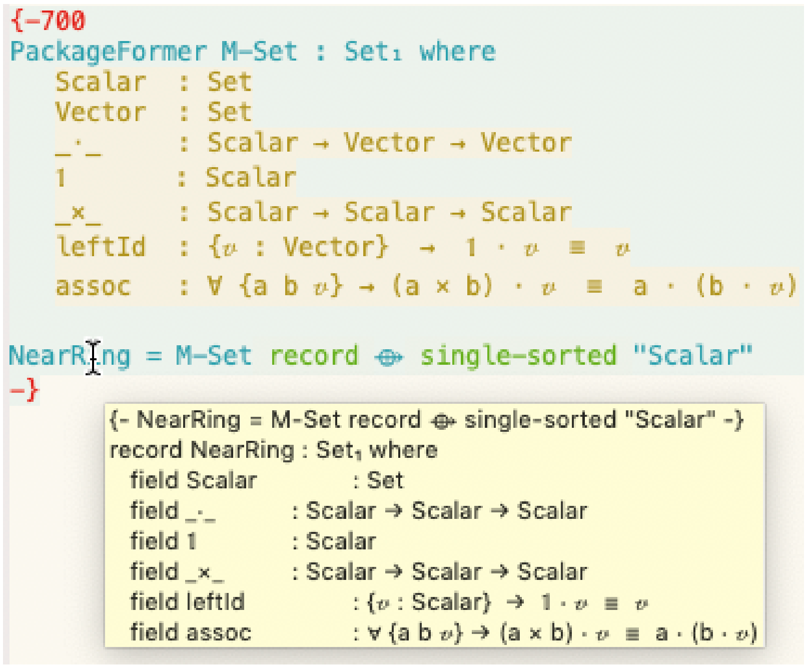
```
{- LeftUnitalSemigroup = M-Set ⟴ monoidal -}
record LeftUnitalSemigroup : Set₁ where
    field Carrier       : Set
    field _⨾_       : Carrier → Carrier → Carrier
    field 𝟙     : Carrier
    field leftId        : {𝑣 : Carrier}  →  𝟙 ⨾ 𝑣  ≡  𝑣
    field assoc     : {a b : Carrier} {𝑣 : Carrier} → (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 Figure 4.3, the source file is furnished with tooltips displaying the 700-comment that a name is associated with, as well as the full elaboration into legitimate Agda syntax. In addition, the above generated elaborations also document the 700-comment that produced them. Moreover, since the editor extension results in valid code in an auxiliary file, future users of a library need not use the `PackageFormer` extension at all —thus we essentially have a static **editor tactic** similar to Agda's (Emacs interface) proof finder.

4.3 Practicality

Herein we demonstrate how to use this system from the perspective of *library designers*. That is to say, we will demonstrate how common desirable features encountered "in the wild" —chapter 3— can be used with our system. The exposition here follows section 2 of the *Theory Presentation Combinators* Carette and O'Connor [CO12], reiterating many the ideas therein. These features are **not built-in** but instead are constructed from a small set of meta-primitives, just as a small core set of language features give way to complex software programs. Moreover, user may combine the meta-primitives —using Lisp— to **extend** the system to produce grouping mechanisms for any desired purpose.

The few constructs demonstrated in this section not only create new grouping mechanisms from old ones, but also create maps from the new, child, presentations to the old parent



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

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

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

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

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 meta-primitives —but we have implemented a few to show that the meta-primitives are deserving of their name.

Do-it-yourself Extendability

In order to make the editor extension immediately useful, and to substantiate the claim that **common module combinators can be defined using the system**, we have implemented a few notable ones, as described in Table 4.1. The implementations, in the user manual, are discussed along with the associated Lisp code and use cases.

Below, in Table 4.2, are the **five meta-primitives** from which all variationals are borne, followed by two others that are useful for extending the system by making your own grouping mechanisms and operations on them. Using these requires a small amount of Lisp.

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

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

Table 4.1: Summary of Sample Variationals Provided With The System

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.

Table 4.2: Metaprogramming Meta-primitives for Making Modules

valued operator,

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

With this perspective, the *sequencing variational combinator* ‘⟴’ is essentially forward function composition/application. Details can be found on the associated webpage; whereas the next chapter provides an Agda function-based semantics.

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

4.3.1 Extension

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

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

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

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

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

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

Economically declaring only the new additions to 'Monoid'

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

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

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

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

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

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

Transport

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

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

4.3.2 Defining a Concept Only Once

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

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

[4]The method call `(s-replace old new s)` replaces all occurrences of string `old` by `new` in the given string `s` ; whereas `(pcase e` $(\mathtt{x}_0\ \mathtt{y}_0)$ $\boxed{\ldots}$ $(\mathtt{x}_n\ \mathtt{y}_n)$`)` pattern matches on `e` and performs the first $\mathtt{y}_i$ if `e =` $\mathtt{x}_i$, otherwise it returns `nil`.

The 'postulating' variational

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

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

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

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

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

Lisp Syntax

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

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

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

Example Use

```
{-700
PackageFormer Magma : Set₁ where
  Carrier : Set
  _·_       : Carrier → Carrier → Carrier

RawRelationalMagma = Magma extended-by "_≈_ : Carrier → Carrier → Set" ⟴ record

RelationalMagma    = RawRelationalMagma postulating "_·_" "congruence" :using "_≈_"
 ↪   ⟴ record
-}
```

Associated Elaboration

```
record RawRelationalMagma : Set₁ where
    field Carrier       : Set
    field op        : Carrier → Carrier → Carrier
    toType      : let View X = X in View Type ; toType = record {Carrier = Carrier}
    field _≈_       : Carrier → Carrier → Set
    toMagma     : let View X = X in View Magma ;    toMagma = record {Carrier =
 ↪   Carrier;op = op}

record RelationalMagma : Set₁ where
    field Carrier       : Set
    field op        : Carrier → Carrier → Carrier
    toType      : let View X = X in View Type ; toType = record {Carrier = Carrier}
    field _≈_       : Carrier → Carrier → Set
    toMagma     : let View X = X in View Magma ;    toMagma = record {Carrier =
 ↪   Carrier;op = op}
    field cong      : ∀ x x' y y' → _≈_ x x' → _≈_ y y' → _≈_ (op x x') (op y
 ↪   y')
    toRawRelationalMagma        : let View X = X in View RawRelationalMagma ;
 ↪   toRawRelationalMagma = record {Carrier = Carrier;op = op;_≈_ = _≈_}
```

(The `let View X = X in View` $\boxed{\cdots}$ clauses are a part of the user implementation of `extended-by`; they are used as markers to indicate that a declaration is a *view* and so should not be an element of the current view constructed by a call to `extended-by`.)

Hence, we have a formal approach to the idea that **each piece of mathematical knowledge should be formalised only once** [GS10].

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

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

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

4.3.3 Renaming

From an end-user perspective, our `CommutativeMonoid` has one flaw: Such monoids are frequently written *additively* rather than multiplicatively. Such a change can be rendered conveniently:

Renaming Example

```
{-700
AbealianMonoidR = CommutativeMonoidR renaming "_·_ to _+_"
-}
```

An Abealian monoid is *both* a commutative monoid and also, simply, a monoid. The above declaration freely maintains these relationships: The resulting record comes with a new projection `toCommutativeMonoidR`, and still has the *inherited* projection `toMonoidR`.

There are a few reasonable properties that a renaming construction should support. Let us briefly look at the properties of `renaming`.

Relationship to Parent Packages

Dual to `extended-by` which can construct (retract) views **to parent** modules mechanically, `renaming` constructs (coretract) views **from parent** packages. That is, it has an optional argument `:adjoin-coretract` which can be provided with `t` to use a default name or provided with a string to use a desired name for the inverse part of a projection, `fromMagma` below.

Adjoining coretracts —views from parent packages

```
{-700
Sequential = Magma renaming "op to _⨾_" :adjoin-coretract t
-}
```

Elaboration

```
record Sequential : Set₁ where
    field Carrier : Set
    field _⨾_     : Carrier → Carrier → Carrier

    toType : let View X = X in View Type
    toType = record {Carrier = Carrier}

    toMagma : let View X = X in View Magma
    toMagma = record {Carrier = Carrier;op = _⨾_}

    fromMagma : let View X = X in Magma → View Sequential
    fromMagma = λ g227742 → record {Carrier = Magma.Carrier g227742;_⨾_ = Magma.op
 ↪  g227742}
```

As the elaboration show, the user implementation of `renaming` makes use of *gensym*'s —generated symbolic names, "fresh variable names"— for λ-arguments to avoid name clashes.

Commutativity

Since `renaming` and `extended-by` (including `postulating`) both adjoin retract morphisms, by default, we are lead to wonder about the result of performing these operations in sequence 'on the fly', rather than naming each application. Since `P renaming X ⟴ postulating Y` comes with a retract `toP` via the `renaming` and another, distinctly defined, `toP` via `postulating`, we have that the operations commute if *only* the first permits the creation of a retract. Below is a concrete example wherein we may replace

```
renaming "_·_ to _⊔_" ⟴ postulating "_⊔_" "idempotent"
```

with

```
postulating "_⊔_" "idempotent" ⟴ renaming "_·_ to _⊔_"
```

and still end up with the same elaboration, up to order of constituents.

```
{-700
IdempotentMagma  = Magma renaming "_·_ to _⊔_" ⟴ postulating "_⊔_" "idempotent"
 ↪   :adjoin-retract nil ⟴ record
-}
```

It is important to realise that the renaming and postulating combinators are *user-defined*, and could have been defined without adjoining a retract by default; consequently, we would have **unconditional commutativity of these combinators**. The user can make these alternative combinators as follows:

Alternative 'renaming' and 'postulating' —with an example use

```
{-700
𝒱-renaming' by = renaming 'by :adjoin-retract nil
𝒱-postulating' p bop (using) = postulating 'p 'bop :using 'using :adjoin-retract
 ↪   nil

IdempotentMagma'' = Magma postulating' "_⊔_" "idempotent" ⟴ renaming' "_·_ to _⊔_"
 ↪   ⟴ record
-}
```

Simultaneous Textual Substitution

As expected, simultaneous renaming works too.

```
{-700
PackageFormer Two : Set₁ where
  Carrier : Set
  𝟘       : Carrier
  𝟙       : Carrier

TwoR = Two record ⟴ renaming' "𝟘 to 𝟙; 𝟙 to 𝟘"
-}
```

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

Involution; self-inverse

Finally, renaming is an invertible operation —ignoring the adjoined retracts, `Magma`^{rr} is identical to `Magma`.

```
{-700
Magmaʳ  = Magma  renaming "_·_  to op"
Magmaʳʳ = Magmaʳ renaming "op   to _·_"
-}
```

Do-it-yourself

Finally, to demonstrate the accessibility of the system, we show how a generic renaming operation can be defined swiftly using the extended set of meta-primitives mentioned in the lower part of Table 4.2. Instead of `renaming` elements *one at a time*, suppose we want to be able to uniformly **rename** all elements in a package. That is, given a function `f` on strings, we want to map over the name component of each element in the package. This is easily done with the following declaration.

Tersely forming a new variational

```
𝒱-rename f = map (λ element → (map-name (λ nom → (funcall f nom))) element)
```

Perhaps the main point of the above definition that may be unexpected to the Agda programmer is that Lisp function calls are of the form (`function` $\texttt{arg}_0$ $\texttt{arg}_1$ ... $\texttt{arg}_n$).

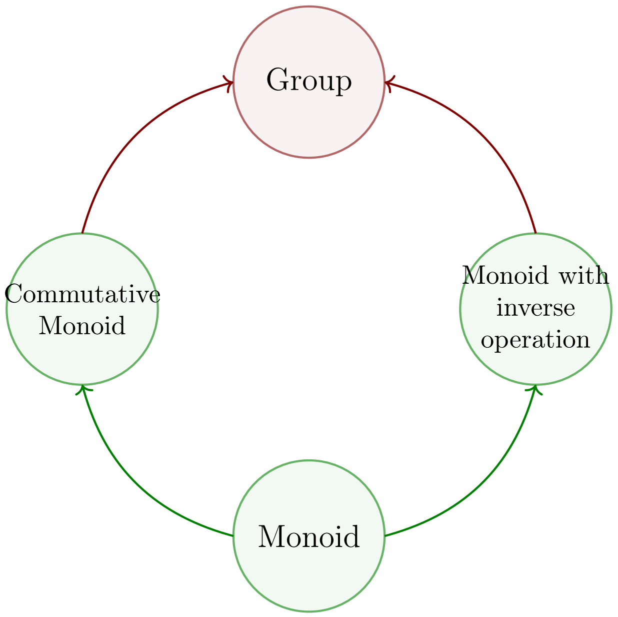
4.3.4 Unions/Pushouts (and intersections)

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

```
{-700
CommutativeGroupR₀ = GroupR extended-by "comm : {x y : Carrier} →  x · y  ≡  y ·
 ↪  x"⟴ record
-}
```

This is **problematic**: We lose the *relationship* that every commutative group is a commutative monoid. This is not an issue of erroneous hierarchical design: From `Monoid`, we could orthogonally add a commutativity property or inverse operation; $\texttt{CommutativeGroupR}_0$ then closes this diamond-loop by adding both features, as shown in Figure 4.3.4. The simplest way to share structure is to union two presentations:

Figure 4.4: Given green, require red



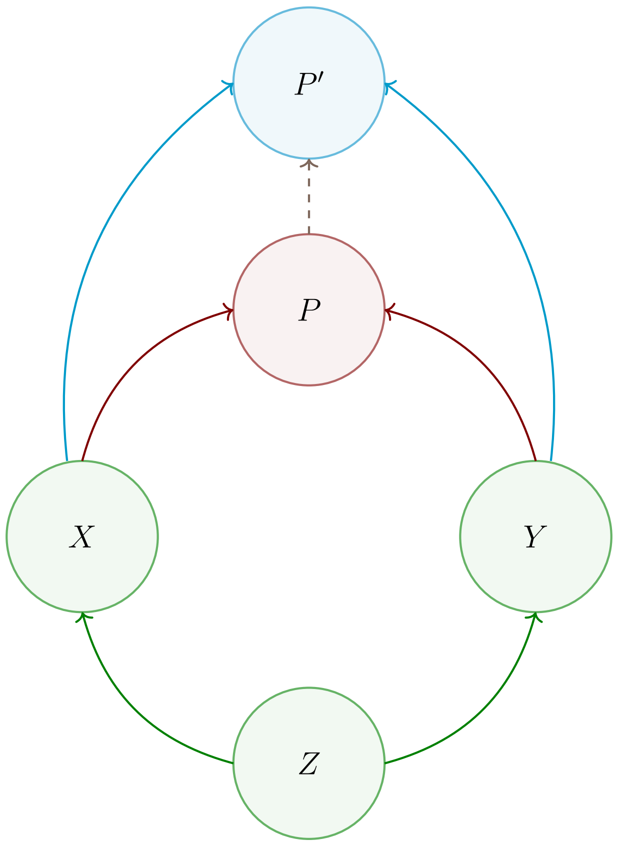
Unions of packages

```
{-700
CommutativeGroupR = GroupR union CommutativeMonoidR ⟴ record
-}
```

The resulting record, `CommutativeMonoidR`, comes with three derived fields —`toMonoidR`, `toGroupR`, `toCommutativeMonoidR`— that retain the results relationships with its hierarchical construction. This approach "works" to build a sizeable library, say of the order of 500 concepts, in a fairly economical way Carette and O'Connor [CO12]. The union operation is an instance of a *pushout* operation, which consists of 5 arguments —three objects and two morphisms— which may be included into the `union` operation as optional keyword arguments. The more general notion of pushout is required if we were to combine `GroupR` with `AbealianMonoidR`, which have non-identical syntactic copies of `MonoidR`.

The pushout of $f : Z \to X$ and $g : Z \to Y$ is, essentially, the disjoint sum of X and Y where embedded elements are considered 'indistinguishable' when they share the same origin in Z via the paths f and g —the pushout generalises the notion of *least upper bound* as shown in Figure 4.3.4 by treating each '$\to$' as a '$\leq$'. Unfortunately, the resulting 'indistinguishable' elements $f(z) \approx g(z)$ are **actually distinguishable**: They may be the f-name or the g-name and a choice must be made as to which name is preferred since users actually want to refer to them later on. Hence, to be useful for library construction, the pushout construction actually requires at least another input function that provides canonical names to the supposedly 'indistinguishable' elements.

Figure 4.5: Given green, require red, such that every candidate cyan has a unique umber



Since a `PackageFormer` is essentially just a *signature* —a collection of typed names—, we can make a 'partial choice of pushout' to reduce the number of arguments from 6 to 4 by letting the typed-names object Z be 'inferred' and encoding the canonical names function into the operations f and g. The inputs functions f, g are necessarily *signature morphisms* —mappings of names that preserve types— and so are simply lists associating names of Z to names of X and Y. If we instead consider $f' : Z' \leftarrow X$ and $g' : Z' \leftarrow Y$, in the *opposite direction*, then we may reconstruct a pushout by setting Z to be common image of f', g', and set f, g to be inclusions. In-particular, the full identity of Z' is not necessarily relevant for the pushout reconstruction and so it may be omitted. Moreover, the issue of canonical names is resolved: *If $x \in X$ is intended to be identified with $y \in Y$ such that the resulting element has z as the chosen canonical name, then we simply require $f'\,x = z = g'\,y$.* An example is shown below in Figure 4.3.4.

At first, a pushout construction needs 5 inputs, to be practical it further needs a function for canonical names for a total of 6 inputs. However, a pushout of $f : Z \to X$ and $g : Z \to Y$ is intended to be the 'smallest object P that contains a copy of X and of Y sharing the common substructure X', and as such it outputs two functions $inj_1 : X \to P$, $inj_2 : Y \to P$ that inject the names of X and Y into P. If we realise P as a record —a type of models— then the embedding functions are *reversed*, to obtain projections $P \to X$ and $P \to Y$: If we have a model of P, then we can forget some structure and rename via f and g to obtain models of X and Y. For the resulting construction to be useful, these names could be automated such as $toX : P \to X$ and $toY : P \to Y$ but such a naming scheme does not scale —but we shall use it for default names. As such, we need two more inputs to the pushout construction so the names of the resulting output functions can be used later on. *Hence, a practical choice of pushout needs 8 inputs!*

Using the above issue to reverse the directions of f, g via f', g', we can infer the shared structure Z and the canonical name function. Likewise, by using $toChild : P \to Child$ default-naming scheme, we may omit the names of the retract functions. If we wish to rename these retracts or simply omit them altogether, we make the *optional* arguments: Provide `:adjoin-retract`$_i$ `"new-function-name"` to use a new name, or `nil` instead of a string to omit the retract —as was done for `extended-by` earlier.

(Abridged) Pushout combinator with 6 optional arguments

```
(𝒱 union pf (renaming₁ "") (renaming₂ "") (adjoin-retract₁ t) (adjoin-retract₂ t)

 = "Union the elements of the parent PackageFormer with those of
    the provided PF symbolic name, then adorn the result with two views:
    One to the parent and one to the provided PF.

    If an identifer is shared but has different types, then crash.

    ADJOIN-RETRACTᵢ, for i : 1..2, are the optional names of the resulting
    views. Provide NIL if you do not want the morphisms adjoined.
    "
   :alter-elements (λ es →
     (let* ((p (symbol-name 'pf))
            (es₁ (alter-elements es renaming renaming₁ :adjoin-retract nil))
            (es₂ (alter-elements ($elements-of p) renaming renaming₂
                                 :adjoin-retract nil))
            (es' (-concat es₁ es₂))
            (name-clashes (loop for n in (find-duplicates (mapcar #'element-name
↪   es'))
                                for e = (--filter (equal n (element-name it)) es')
                                unless (--all-p (equal (car e) it) e)
                                collect e))
            (er₁ (if (equal t adjoin-retract₁) (format "to%s" $parent)
                   adjoin-retract₁))
            (er₂ (if (equal t adjoin-retract₂) (format "to%s" p)
                   adjoin-retract₂)))

       ;; Error on name clashes; unabridged version has a mechanism to “fix
↪   conflicts”
       ;; The unabridged version accounts for name clashes on retracts as well.
       (if name-clashes
            (-let [debug-on-error nil]
              (error "%s = %s union %s \n\n\t\t → Error:
                      Elements ‘‘%s’’ conflict!\n\n\t\t\t%s"
                      $name $parent p (element-name (caar name-clashes))
                      (s-join "\n\t\t\t" (mapcar #'show-element (car
↪   name-clashes))))))

   ;; return value
   (-concat es'
            (and adjoin-retract₁ (not er₁) (list (element-retract $parent es :new
↪   es₁ :name adjoin-retract₁)))
            (and adjoin-retract₂ (not er₂) (list (element-retract p ($elements-of p)
↪   :new es₂ :name adjoin-retract₂)))))))
```

The reader is not meant to understand the (abridged[5]) definition provided here, however we

[5]The unabridged definition, on the `PackageFormer` webpage, has more features. In particular, it accepts additional keyword toggles that dictate how it should behave when name clashes occur; e.g., whether it should halt and report the name clash or whether it should silently perform a name change, according to another provided argument. The additional flexibility is useful for rapid experimentation.

present a few implementation remarks and wish to emphasise that this definition is **not built in**, and so the user could have, for example, provided a faster implementation by omitting checks for name clashes.

1. Since the systems allows optional keyword arguments, the first line declares only a context name, `pf`, is mandatory and the remaining arguments to a pushout are 'inferred' unless provided.
2. The second line documents this new user-defined variational; the documentation string is attached as a tooltip to all instances of the phrase `union`.
3. Given `f, g` as `renaming`$_i$, we apply the renaming variational on the elements of the implicit context (to this variational) and to the given context `pf` to obtain two new element lists `e`$_i$.
4. We then adjoin retract elements `er`$_i$.
5. Finally, we check for name clashes and handle them appropriately.

The user manual contains full details and an implementation of intersection, pullback, as well. We now turn to some examples of this construction to see it in action; in particular, here is an example which mentions all arguments, optional and otherwise. Besides the specification's elaboration, we also provide a **commutative** diagram, Figure 4.3.4, that *informally* carries out the `union` construction.

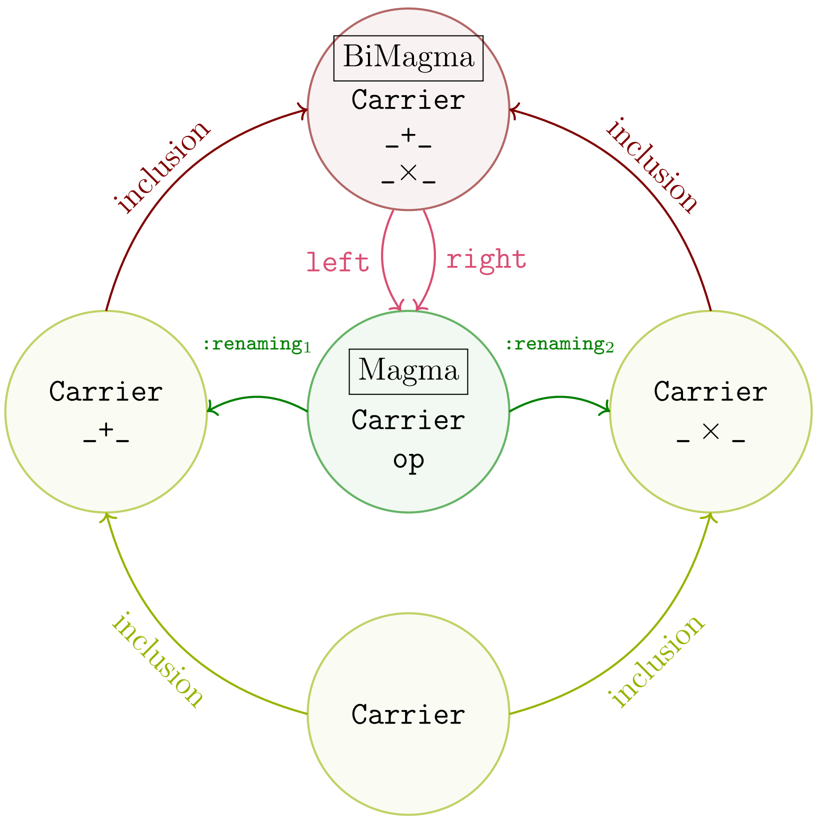
Bimagmas: Two magmas sharing the same carrier

```
{-700
TwoBinaryOps = Magma union Magma :renaming₁ "op to _+_" :renaming₂ "op to _×_"
 ↪   :adjoin-retract₁ "left" :adjoin-retract₂ "right"
-}
```

Elaboration

```
record TwoBinaryOps : Set₁ where
    field Carrier : Set
    field _+_     : Carrier → Carrier → Carrier

    toType : let View X = X in View Type
    toType = record {Carrier = Carrier}

    field _×_     : Carrier → Carrier → Carrier

    left : let View X = X in View Magma
    left = record {Carrier = Carrier;op = _+_}

    right : let View X = X in View Magma
    right = record {Carrier = Carrier;op = _×_}
```

Figure 4.6: Given green, yield yellow, require red, form fuchsia



Remember, *this particular user implementation* realises

$$\texttt{X}_1 \texttt{ union } \texttt{X}_2 \texttt{ :renaming}_1 \texttt{ f}\boxed{\texttt{'}} \texttt{ :renaming}_2 \texttt{ g}\boxed{\texttt{'}}$$

as the pushout of the inclusions $\texttt{f}\boxed{\texttt{'}}\ \texttt{X}_1\ \boxed{\cap}\ \texttt{g}\boxed{\texttt{'}}\ \texttt{X}_2\ \boxed{\hookrightarrow}\ \texttt{X}_i$ where the source is the set-wise intersection of *names*. Moreover, when either $\texttt{renaming}_i$ is omitted, it defaults to the identity function.

We now turn to useful properties of the user-defined `union` variational.

Idempotence —Set Union

The next example is one of the reasons the construction is named ‘union’ instead of ‘pushout’: It’s idempotent, if we ignore the addition of the retract.

```
{-700
MagmaAgain    = Magma union Magma
-}
```

```
record MagmaAgain : Set₁ where
    field Carrier : Set
    field op      : Carrier → Carrier → Carrier

    toType : let View X = X in View Type
    toType = record {Carrier = Carrier}

    toMagma : let View X = X in View Magma
    toMagma = record {Carrier = Carrier;op = op}
```

Of note is that this is essentially the previous bi-magma example —Figure 4.3.4— *but* we are not distinguishing —via $\texttt{:renaming}_i$— the two instances of `Magma`.

Disjointness —Categorial Sums

We may perform disjoint sums —simply distinguish all the names of one of the input objects.

```
{-700
Magma’    = Magma primed ⟴ record
SumMagmas = Magma union Magma’ :adjoin-retract₁ nil ⟴ record
-}
```

Elaboration

```
record SumMagmas : Set₁ where
    field Carrier  : Set
    field op       : Carrier → Carrier → Carrier

    toType         : let View X = X in View Type
    toType = record {Carrier = Carrier}

    field Carrier’ : Set
    field op’      : Carrier’ → Carrier’ → Carrier’

    toType’ : let View X = X in View Type
    toType’ = record {Carrier = Carrier’}

    toMagma : let View X = X in View Magma
    toMagma = record {Carrier = Carrier’;op = op’}

    toMagma’ : let View X = X in View Magma’
    toMagma’ = record {Carrier’ = Carrier’;op’ = op’}
```

Of note is that this is essentially the previous bi-magma example —Figure 4.3.4— *but* we are not distinguishing the two instances of `Magma` ‘on the fly’ via `:renaming`$_i$ but instead making them disjoint beforehand using the following *informal* equation:

$$\texttt{p primed} \;\boxed{\approx}\; \texttt{p :renaming (}\lambda\texttt{ name} \boxed{\to} \texttt{name ++ "’")}$$

Support for Diamond Hierarchies

A common scenario is extending a structure, say `Magma`, into orthogonal directions, such as by making it operation associative or idempotent, then closing the resulting diamond by combining them, to obtain a semilattice. However, the orthogonal extensions may involve different names and so the resulting semilattice presentation can only be formed via pushout; below are three ways to form it.

Three ways to get to SemiLattice

```
{-700
Semigroup          = Magma postulating "_·_" "associative"
IdempotentMagma    = Magma renaming "_·_ to _⊔_" ⟴ postulating "_⊔_" "idempotent"
 ↪  :adjoin-retract nil

⊔-SemiLattice     = Semigroup union IdempotentMagma :renaming₁ "_·_ to _⊔_"
·-SemiLattice     = Semigroup union IdempotentMagma :renaming₂ "_⊔_ to _·_"
↑-SemiLattice     = Semigroup union IdempotentMagma :renaming₁ "_·_ to _↑_"
 ↪  :renaming₂ "_⊔_ to _↑_"
-}
```

Application: Granular (Modular) Hierarchy for Rings

We will close with the classic example of forming a ring structure by combining two monoidal structures. This example also serves to further showcase how using `postulating` can make for more granular, modular, developments.

```
{-700
Additive           = Magma renaming "_·_ to _+_" ⟴ postulating "_+_" "commutative"
 ↪  :adjoin-retract nil ⟴ record
Multiplicative     = Magma renaming "_·_ to _×_" :adjoin-retract nil ⟴ record
AddMult            = Additive union Multiplicative ⟴ record
AlmostNearSemiRing = AddMult ⟴ postulating "_×_" "distributiveˡ" :using "_+_" ⟴
 ↪  record
-}
```

Elaboration

```
record AlmostNearSemiRing : Set₁ where
    field Carrier : Set
    field _+_     : Carrier → Carrier → Carrier

    toType : let View X = X in View Type
    toType = record {Carrier = Carrier}

    toMagma : let View X = X in View Magma
    toMagma = record {Carrier = Carrier;op = _+_}

    field comm       : ∀ x y   → _+_ x y ≡ _+_ y x
    field _×_         : Carrier → Carrier → Carrier

    toAdditive : let View X = X in View Additive
    toAdditive = record {Carrier = Carrier;_+_ = _+_;comm = comm}

    toMultiplicative : let View X = X in View Multiplicative
    toMultiplicative = record {Carrier = Carrier;_×_ = _×_}

    field distˡ      : ∀ x y z → _×_ x (_+_ y z) ≡ _+_ (_×_ x y) (_×_ x z)
```

4.3.5 Duality

Maps between grouping mechanisms are sometimes called *views*, which are essentially an internalisation of the *variationals* in our system. A useful view is that of capturing the heuristic of *dual concepts*, e.g., by changing the order of arguments in an operation. That is, the dual, or opposite, of a binary operation `_·_ : X → Y → Z` is the operation $\_\cdot^{op}\_$ `: Y → X → Z` defined by x $\cdot^{op}$ y = y · x. Classically in Agda, duality is *utilised* as follows:

1. Define a *parameterised* module R $\_\boxed{\cdot}\_$ for the desired ideas **on** the operation $\_\boxed{\cdot}\_$.

   ◇ Concretely, say it defines the predicate `·-isLeftId e = (∀ x → e · x ≡ x)`.

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

   ◇ Continuing the concrete example, R^{op} $\_\boxed{\cdot}\_$ would essentially be

```
public open R _·_ renaming (·-isLeftId to ·-isRightId)
```

The Ubiquity of Duality

The RATH-Agda [Kah18] library performs essentially this approach, for example for obtaining `UpperBounds` from `LowerBounds` in the context of an ordered set. Moreover, since Category Theory can serve as a foundational system of reasoning (logic) and implementation (programming), the idea of duality immediately applies to produce “two for one” theorems and programs.

Unfortunately, this means that any record definitions in `R` must have their field names be sufficiently generic to play *both* roles of the original and the dual concept. Admittedly, RATH-Agda’s names are well-chosen; e.g., `value`, `bound`$_i$, `universal` to denote a `value` that is a lower/upper `bound` of two given elements, satisfying a least upper bound or greatest lower bound `universal` property. However, well-chosen names come at an upfront cost: One must take care to provide sufficiently generic names and account for duality at the outset, irrespective of whether one *currently* cares about the dual or not; otherwise when the dual is later formalised, then the names of the original concept must be refactored throughout a library and its users. This is not the case using `PackageFormer`. Consider the following heterogeneous algebra —which is essentially the main example of section 4.2 but missing the associativity field.

Left unital actions

```
{-700
PackageFormer LeftUnitalAction : Set₁ where
  Scalar : Set
  Vector : Set
  _·_     : Scalar → Vector → Vector
  𝟙       : Scalar
  leftId  : {x : Vector} → 𝟙 · x ≡ x

-- Let’s reify this as a valid Agda record declaration
LeftUnitalActionR  = LeftUnitalAction ⟴ record
-}
```

Informally, one now ‘defines’ a right unital action by duality, flipping the binary operation and renaming `leftId` to be `rightId`. Such informal parlance is in-fact nearly formally, as the following:

Right unital actions —mechanically by duality

```
{-700
RightUnitalActionR = LeftUnitalActionR flipping "_·_" :renaming "leftId to rightId"
 ↪   ⟴ record
-}
```

Of-course the resulting representation is semantically identical to the previous one, and so it is furnished with a *toParent* mapping:

```
forget : RightUnitalActionR → LeftUnitalActionR
forget = RightUnitalActionR.toLeftUnitalActionR
```

Likewise, for the RATH-Agda library's example from above, to define semi-lattice structures by duality:

```
import Data.Product as P

{-700
PackageFormer JoinSemiLattice : Set₁ where
  Carrier : Set
  _⊑_     : Carrier → Carrier → Set

  refl    : ∀ {x}     → x ⊑ x
  trans   : ∀ {x y z} → x ⊑ y → y ⊑ z → x ⊑ z
  antisym : ∀ {x y}   → x ⊑ y → y ⊑ x → x ≡ y

  _⊔_     : Carrier → Carrier → Carrier
  ⊔-lub   : ∀ {x y z} → x ⊑ z → y ⊑ z → (x ⊔ y) ⊑ z
  ⊔-lub˘  : ∀ {x y z} → (x ⊔ y) ⊑ z  →  x ⊑ z  P.×  y ⊑ z

JoinSemiLatticeR = JoinSemiLattice record
MeetSemiLatticeR = JoinSemiLatticeR flipping "_⊑_" :renaming "_⊔_ to _⊓_; ⊔-lub to
 ↪  ⊓-glb"
-}
```

In this example, besides the map from meet semi-lattices to join semi-lattices, the types of the dualised names, such as `⊓-glb`, are what one would expect were the definition written out explicitly:

Checking the types of the duals

```
module woah (M : MeetSemiLatticeR) where
  open MeetSemiLatticeR M

  lub_dual_type : ∀ {x y z} → z ⊑ x → z ⊑ y → z ⊑ (x ⊓ y)
  lub_dual_type = ⊓-glb

  trans_dual_type : let _⊒_ = λ x y → y ⊑ x
                    in ∀ {x y z} → x ⊒ y → y ⊒ z → x ⊒ z
  trans_dual_type = trans
```

4.3.6 Extracting Little Theories

The `extended-by` variational allows Agda users to easily employ the *tiny theories* Farmer, Guttman, and Javier Thayer [FGJ92] and Carette et al. [Car+11] 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** [Mar92; FR14]: *No client should be forced to depend on methods it does not use.*

Tiny Theories Example

```
{-700
PackageFormer Empty : Set₁ where {- No elements -}
Type  = Empty extended-by "Carrier : Set"
Magma = Type  extended-by "_·_ : Carrier → Carrier → Carrier"
CommutativeMagma = Magma extended-by "comm : {x y : Carrier} →  x · y  ≡  y · x"
-}
```

However, life is messy and sometimes one may hurriedly create a structure, then later realise that they are being forced to depend on unused methods. Rather than throw a `not implemented` exception or leave them undefined, we may use the `keeping` variational to **extract the smallest well-formed sub-PackageFormer that mentions a given list of identifiers**. For example, suppose we quickly formed `Monoid` **monolithicaly** as presented at the start of section 4.3.1, but later wished to utilise other substrata. This is easily achieved with the following declarations.

Extracting Substrata from a Monolithic Construction

```
{-700
Empty”        = Monoid keeping ""
Type”         = Monoid keeping "Carrier"
Magma”        = Monoid keeping "_·_"
Semigroup”    = Monoid keeping "assoc"
PointedMagma” = Monoid keeping "▯; _·_"
                -- This is just keeping: Carrier; _·_; ▯
-}
```

Even better, we may go about deriving results —such as theorems or algorithms— in familiar settings, such as `Monoid`, only to realise that they are written in **settings more expressive than necessary**. Such an observation no longer need to be found by inspection, instead it may be derived mechanically.

Specialising a result from an expressive setting to the **minimal** necessary setting

```
{-700
LeftUnitalMagma = Monoid keeping "▯-unique" ⟴ record
-}
```

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

Elaboration

```
record LeftUnitalMagma : Set₁ where

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

   ▯-unique     : ∀ {e} (lid : ∀ {x} → e · x ≡ x) (rid : ∀ {x} → x · e ≡ x) → e
 ↪   ≡ ▯
   ▯-unique lid rid = ≡.trans (≡.sym leftId) rid
```

Surprisingly, in some sense, `keeping` let's us apply the interface segregation principle, or 'little theories', **after the fact** —this is also known as *reverse mathematics*.

4.3.7 200+ theories —one line for each

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

In order to demonstrate the **immediate practicality** of the ideas embodied by `PackageFormer`, we have implemented a list of mathematical concepts from universal algebra —which is useful to computer science in the setting of specifications. The list of structures is adapted from the source of a MathScheme library Carette and O'Connor [CO12] and Carette et al. [Car+11], which in turn was inspired by web lists of Peter Jipsen, John Halleck, and many others from Wikipedia and nlab. Totalling over 200 theories which elaborate into nearly 1500 lines of typechecked Agda, this demonstrates that our systems works; the **750% efficiency savings** speak for themselves.

The 200+ one line specifications and their ~1500 lines of elaborated typechecked Agda can be found on `PackageFormer`'s webpage.

`https://alhassy.github.io/next-700-module-systems`

If anything, this elaboration demonstrates our tool as a useful engineering result. The main novelty being the ability for library users to extend the collection of operations on packages, modules, and then have it immediately applicable to Agda, an **executable** programming language.

Since the resulting **expanded code is typechecked** by Agda, we encountered a number of places where non-trivial assumptions accidentally got-by the MathScheme team. For example, in a number of places, an arbitrary binary operation occurred multiple times leading to ambiguous terms, since no associativity was declared. Even if there was an implicit associativity criterion, one would then expect multiple copies of such structures, one axiomatisation for each parenthesisation. Moreover, there were also certain semantic concerns about the design hierarchy that we think are out-of-place, but we chose to leave them as is —e.g., one would think that a "partially ordered magma" would consist of a set, an order relation, and a binary operation that is monotonic in both arguments; however, `PartiallyOrderedMagma` instead comes with a single monotonicity axiom which is only equivalent to the two monotonicity claims in the setting of a monoidal operation. Nonetheless, we are grateful for the source file provided by the MathScheme team.

Extensiblity

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

4.4 Contributions: From Theory to Practice

The `PackageFormer` implements the ideas of Chapters 2 and 3. As such, as an editor extension, it is mostly **language agnostic** and could be altered to work with other languages such as Coq [Chr03], Idris [Bra16], and even Haskell [LM13]. The `PackageFormer` implementation has the following useful properties.

1. Expressive & extendable specification language for the library developer.
    - ◇ Our meta-primitives give way to the ubiquitous module combinators of Table 4.1.
    - ◇ E.g., from a theory we can derive its homomorphism type, signature, its termtype, etc; we generate useful constructions inspired from universal algebra and seen in the wild —see Chapter 3.

- ◇ An example of the freedom allotted by the extensible nature of the system is that combinators defined by library developers can, say, utilise auto-generated names when names are irrelevant, use ‘clever’ default names, and allow end-users to supply desirable names on demand using Lisps’ keyword argument feature —see section 4.3.4.

2. Unobtrusive and a tremendously simple interface to the end user.
   - ◇ Once a library is developed using (the current implementation of) `PackageFormer`, the end user only needs to reference the resulting generated Agda, without any knowledge of the existence of `PackageFormer`.
     - ○ Generated modules are necessarily ‘flattened’ for typechecking with Agda — see section 4.3.1.
   - ◇ We demonstrates how end-users can build upon a library by using *one line* specifications, by reducing over 1500 lines of Agda code to nearly 200 specifications using `PackageFormer` syntax.
3. Efficient: Our current implementation processes over 200 specifications in ~3 seconds; yielding typechecked Agda code *which* is what consumes the majority of the time.
4. Pragmatic: Common combinators can be defined for library developers, and be furnished with concrete syntax for use by end-users.
5. Minimal: The system is essentially invariant over the underlying type system; with the exception of the meta-primitive `:waist` which requires a dependent type theory to express ‘unbundling’ component fields as parameters.
6. Demonstrated expressive power *and* use-cases.
   - ◇ Common boiler-plate idioms in the standard Agda library, and other places, are provided with terse solutions using the `PackageFormer` system.
     - ○ E.g., automatically generating homomorphism types and wholesale renaming fields using a single function —see section 4.3.3.
   - ◇ Over 200 modules are formalised as one-line specifications.
7. Immediately useable to end-users *and* library developers.
   - ◇ We have provided a large library to experiment with —thanks to the MathScheme group for providing an adaptable source file.
   - ◇ In the online user manual, we show how to formulate module combinators using a simple and straightforward subset of Emacs Lisp —a terse introduction to Lisp is provided.

Recall that we alluded —in the introduction to section 4.3— that we have a categorical structure consisting of `PackageFormers` as objects and those variationals that are signature morphisms. While this can be a starting point for a semantics for `PackageFormer`, we will instead pursue a *mechanised semantics.* That is, we shall encode (part of) the syntax of `PackageFormer` as Agda functions, thereby giving it not only a semantics but rather a life in a familar setting and lifting it from the status of *editor extension* to *language library.*

Bibliography

[Bra16] Edwin Brady. *Type-driven Development With Idris*. Manning, 2016. ISBN: 9781617293023. URL: `http://www.worldcat.org/isbn/9781617293023` (cit. on p. 41).

[Car+11] Jacques Carette et al. *The MathScheme Library: Some Preliminary Experiments*. 2011. arXiv: `1106.1862v1 [cs.MS]` (cit. on pp. 39, 40).

[Chr03] Jacek Chrzaszcz. "Implementing Modules in the Coq System". In: *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*. 2003, pp. 270–286. DOI: `10.1007/10930755\_18`. URL: `https://doi.org/10.1007/10930755%5C_18` (cit. on p. 41).

[CO12] 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` (cit. on pp. 15, 27, 40).

[FGJ92] 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 (cit. on p. 39).

[FR14] 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` (cit. on p. 39).

[GS10] 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` (cit. on p. 22).

[Kah18] Wolfram Kahl. *Relation-Algebraic Theories in Agda*. 2018. URL: `http://relmics.mcmaster.ca/RATH-Agda/` (visited on 10/12/2018) (cit. on p. 37).

[LM13] 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` (cit. on p. 41).

[Mar92] 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) (cit. on p. 39).