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

May 10, 2020

PhD Thesis

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

[WK: Please resolve references before you ship PDF. . . **]**

Abstract

Structuring-mechanisms, such as Java's `package` and Haskell's `module`, are often afterthought secondary citizens whose primary purpose is to act as namespace delimiters, while relatively more effort is given to their abstraction encapsulation counterparts, e.g., Java's classes and Haskell's typeclasses. A *dependently-typed language* (DTL) is a typed language where we can write *types* that depend on *terms*; thereby blurring conventional distinctions between a variety of concepts. In contrast, languages with non-dependent type systems tend to distinguish *external vs. internal* structuring-mechanisms —as in Java's `package` for namespacing vs. `class` for abstraction encapsulation— with more dedicated attention and power for the internal case —as it is expressible within the type theory.

To our knowledge, relatively few languages —such as OCaml, Maude, and the B Method— allow for the manipulation of external structuring-mechanisms as they do for internal ones. Sufficiently expressive type systems, such as those of dependently typed languages, allow for the internalisation of many concepts thereby conflating a number of traditional programming notions. Since DTLs permit types that depend on terms, the types may require non-trivial term calculation in order to be determined. Languages without such expressive type systems necessitate certain constraints on its constructs according to their intended usage. It is not clear whether such constraints have been brought to more expressive languages out of necessity or out of convention. Hence we propose a systematic exploration of the structuring-mechanism design space for dependently typed languages to understand *what are the module systems for DTLs?*

First-class structuring-mechanisms have values and types of their own which need to be subject to manipulation by the user, so it is reasonable to consider manipulation combinators for them from the beginning. Such combinators would correspond to the many generic operations that one naturally wants to perform on structuring-mechanisms —e.g., combining them, hiding components, renaming components— some of which, in the external case, are impossible to perform in any DTL without resorting to third-party tools for pre-processing. Our aim is to provide a sound footing for systems of structuring-mechanisms so that structuring-mechanisms become another common feature in dependently typed languages. An important contribution of this work is an Agda implementation of our module combinators —which we hope to be accepted into a future release of the Agda standard library.

If anything, our aim is practical —to save developers from ad hoc copy-paste preprocessing hacks.

—Source: `https://github.com/alhassy/next-700-module-systems`—

Contents

List of Tables

[Editor Remark:

Re-read everything and make sure if anything is 'partly borrowed' from another source then it is properly cited!

"If you knowingly "borrowed" even one clause, let alone one sentence, then you have committed plagiarism. Think of it this way: Plagiarism is another word for theft. That's exactly what it is. Were you to open a bag of bread in a grocery store and eat one piece of bread, leaving the remainder of the bag on the shelf, you would have knowingly stolen that one piece of bread. The fact that you didn't take the rest of the bread with you doesn't negate the theft of that one piece. That one sentence you noted is like that one piece of bread. It's stolen. It wasn't yours. You didn't own it. You took it from someone else. Someone had to write that sentence in order for it to exist. By cutting and pasting it into your "original" work, you committed theft. So, yes, plagiarism even comes down to one sentence." —Ninth Ward Goethe $\boxed{]}$

Chapter 1

Introduction

The composition of heterogeneous components, consisting of various types and values, leads to large-scale, maintainable, software systems. The components are often of a domain-specific nature, packed-up using a module system that exists only in the periphery. The computations therein may themselves involve 'first-class modules', for which there usually is only a limited, non-extensible, number of operators that can be applied —with `new, open, import, using` being the common suspects. The rising popularity of functional programming and dependently-typed languages has contributed to the 'internalisation' of what were once considered 'second-class' citizens in a language. For example, method blocks becomes functional values, and types become values inhabiting larger types. Consequently, this has prompted a search for a minimal number of primitives that provide a rich and expressive language that not only accounts for inter-language operations but also intra-language concepts, such as first-class modules. Therefore, an effort needs to be made in order to design and implement systems which cut-away unnecessary clutter and provide a core kernel that is powerful enough to support existing pragmatic use cases. In particular, *the goal is to use a dependently-typed language to implement the 'missing' module system features directly inside the language.*

This chapter introduces the context and problem domain of this thesis and motivates the need for a framework for a first-class treatment of modules in dependently typed languages. More precisely, Section 1.1 gives an overview of dependently-typed languages and Section 1.2 discusses the burdens of moving to expressive type systems. Section 1.3 provides a shallow motivation for the internalising of modules as a natural progression of dependently-typed languages. Section 1.4 demonstrates the interdefinability of three prominent structuring mechanisms with the context of a dependently-typed language, thereby outlining the specific context of this thesis. Section 1.5 states the proposed research problem, outlines the objectives of this thesis, and discusses the highlights of the approach taken to achieve each objective. Section 1.6 summarises the contributions of this thesis to the enhanced understanding of modular module systems within dependently-typed languages. Section 1.7 notes the publications related to the work presented in this thesis. Finally, Section 1.8 outlines the structure of the remainder of this thesis.

1.1 Overview of Dependently-Typed Languages (DTLs)

Software systems typically consist of numerous components whose constituent data may be parameterised or accessed without qualifiers or have a corresponding descriptive syntax type for serialisation and metaprogramming. Ensuring that the diverse variations on a component of grouped data —parameterised, namespace, description— remain in sync is a challenging task. Generally speaking, this thesis explores the area of library development and focuses on two issues: *Write once, generate many* approach to modules, and *a practical interface for module users* to 'hoist up' module constituents as opaque parameters which can then be fixed 'once and for all' to provide a new product type.

Dependent-types allow us to encode properties of data *within the structure* of the data itself, and so all the data we consider is necessarily 'well-formed'. In contrast, without dependent types, one would (1) declare a data structure, *then* (2) define the subclass of such data that is 'well-formed' in some sense; *then*, (3) to work with this data, one provides an interface that only produces well-formed data, a so-called 'smart constructor', *finally*, one needs to test that their smart constructor actually only forms well-defined data elements. For instance, raw untyped λ-terms are not all sensible, and so one introduces types to organise them into sensible classes, then introduces inference rules that ensure only sensible terms are constructed.

> DTLs flatten the conventional four-stage process of declaring raw data, selecting a coherent subclass, providing a smart constructor, and proving the constructor is valid.

We shall explain this idea more concretely via two examples, below in Sections 1.1.2, 1.1.3. The Agda fragments presented will be explained in the accompanying text —an introduction to Agda is given in Appendix B. Afterword, we conclude by briefly mentioning theoretical concerns when working with DTLs and, more importantly for topic on modularisation, issues of a more practical nature involving library development.

Types	Machine check-able 'comments'; coherent expressions
Polymorphism	Uniform definitions; avoiding repetition
Dependent types	Uniform treatment of values and types, section 1.1.1 and increased expressivity of 'comments'

Table 1.1: Why we are interested in DTLs?

The above table tersely summarises our desire for powerful type systems. In particular, type polymorphism permits us to produce functions written once with type variables and have them applied to radically different types. Likewise, it would be desirable to write once a generic function on a kind of package and have it operate on the many variations of packaging. An example of this idea is presented in Section 4.2.

1.1.1 Uniformity

A type alias and a value alias are merely aliases at the end of the day, so unlike Haskell, for example, which distinguishes the two, Agda, for example, does not. More generally, type families, simple types, type constructors, dependent types, etc, collapse into a single category: Dependent types.

In particular, recall the canonical definition of 'term':

Grammar for Terms

```
term ::=  x                     {- variable             -}
      |   f(term₀, ..., termₙ)  {- function application -}
```

In pedestrian languages, one distinguishes between *value* terms and *type* terms, whence the `term`$_i$ are constrained to be homogeneously all values or all types. In contrast, a dependently-typed languages makes no such limitation, thereby allowing the `term`$_i$ to be heterogeneous. For example, in a simple type system, `Maybe (A × List B)` is a term where all variables, `term`$_0$`, term`$_1$` = A, B`, are of the same kind —types. This is not so with the term[1] `Maybe (A × Vec B n)` —`A` and `B` are types while `n` is a number. This is the essence of DTLs, and a primary reason we want to use them.

In the same vein, the varying notions of packaging are treated differently even though they are isomorphic in certain scenarios or interdefinable in others. As such, it would be useful to reduce the syntactic distinction between them.

1.1.2 Example 1: Sanitising raw data

When interacting with users, a system receives raw data then 'sanitises' it, or ensures it is 'sanitised'. For instance, to subscribe to a mailing list, a user provides a string of symbols which the program then ensures is a well-formatted email address. Below is a possible implementation of the email address portion within Haskell —the comments are a designer's thought process as *allowed* by the coding language.

[1]The standard type `Vec A n`, consisting of lists of elements of `A` having length `n`, is defined in Appendix B.

Traditional Four Stages to Structuring Data

```
{- (1) An email address is just a raw string -}
data Email = MkEmail String  deriving Show

{- (2) Actually, it has some structure -}
isValid :: Email -> Bool
isValid (MkEmail s) = let pre_rest = splitOn "@" s
                      in length pre_rest == 2
                      && length (splitOn ".com" (pre_rest !! 1)) == 1

{- (3) Given two strings, we can form an email address -}
mkEmail :: String -> String -> Email
mkEmail pre post = MkEmail (pre ++ "@" ++ post ".com")

{- (4) Also, mkEmail is a smart constructor for Email -}
{- ∀ pre post • isValid (mkEmail pre post)        -}
```

With dependent types, we can *encode* structural[2] properties: We can declare a type of strings necessarily of the form ⟨`string`⟩`@`⟨`string`⟩`.com`, thereby dispensing with any sanitation phase. In particular, in this style, a parser is essentially a type-checker. Moreover such checks happen at compile time since these are just like any other type.

Parsing ≈ Typechecking

```
data Email : String → Set where
  MkEmail : (pre post : String) → Email (pre ++ "@" ++ post ++ ".com")
```

The above declaration defines a new type `Email s` with values `MkEmail pre post` *precisely when* `s` ≈ `pre ++ "@" ++ post ++ ".com"`. Hence, any value of `Email s` is, by its very construction, a pair of strings, say, `pre` and `post` that compose to give the original address `s`. The above four steps in Haskell have been reduced to a single declaration in Agda.

What happened exactly? Where are the dependent-types? Let `X` denote the type of strings, `Y` the type of pairs of strings, `P` the property "x is composed of the pair y", and the lower-case `p` is the proviso in the Haskell code above. Let $\mathcal{Y}$ absorp the proviso property `p` —in the Agda code, this amounts to "building `p` into the type"— so that `y` ∈ $\mathcal{Y}$`(x)` ≡ `p(x, y)`. The type $\mathcal{Y}$ is a dependent type: It is a type that *depends* on a term; namely, `x`. Then the transition from specification, to Haskell implementation, to Agda code can be summarised in the following chain of equalities.

When claims only hold under certain expected premises, it would be easier to reason and state the claims if such preconditions were incorporated into the types. This is common

[2]Arbitrary, semantic, properties can be attached to data constructors. However, properties encoded via syntactic structure can be mechanically checked via typechecking. Whereas needing *a proof of a property* may require human intervention.

$$
\begin{array}{ll}
 & \textit{Every email address decomposes into a pair of strings} \\
\approx & \forall\ \mathtt{x : X} \bullet \exists\ \mathtt{y : Y} \bullet \mathtt{p(x,\ y)} \land \mathtt{P(x,\ y)} \\
\approx & \forall\ \mathtt{x : X} \bullet \exists\ \mathtt{y} : \mathcal{Y}\mathtt{(x)} \bullet \mathtt{P(x,\ y)}
\end{array}
$$

Table 1.2: Dependent types 'absorp' preconditions

practice in mathematics —e.g., "the maximum operation over real numbers has a least element when *only considering* non-negative whole numbers" versus "the maximum operation *on naturals* has a least element"; i.e., mathematicians *declare a new set* $\backslash\ \mathbb{N} = \{r : \mathbb{R} \mid r \geq 0 \land \lceil r \rceil = r\}$. However, in conventional programming, there is no way to *form such a new type* denoting "the values of type A that satisfy property B"; unless you have access to dependent types, which call this type Σ `a : A` $\bullet$ `B(a)`.

1.1.3 Example 2: Correct-by-Construction Programming

Program verification is an 'after the fact' activity, like documentation; yet when a project behaves as desired, programmers seldom willingly go back to clean up and instead prefer a new project. This dissociation of concerns is remedied by enabling program verification to proceed side-by-side with development Gries [Gri81], Cohen [Coh90], and Dijkstra [Dij76]: Each proof of a program property acts as exhaustive test cases for that property.

With a careful specification of the type, there is only one[3] *program!*

For example, suppose we want an implementation of a function f specified by the property `f 0 = 1` $\land$ `f (n + 1) = n` $\times$ `f n`, for any `n`. The first conjunct completely determines `f` on input `0`, however an inattentive implementer may decide to define \ `f n := f (n + 1) / n`. The resulting 'definition' clearly satisfies the specification, but it does not terminate on any positive input since it recursively calls itself on ever increasing arguments.

In comparison, since Agda requires all its functions to be terminating, after insisting the specification obligations hold by definition, `refl`, we turn to defining `f` by pattern matching and its implementation from there is fully forced: There are no more choices in implementation. Then, Agda's Emacs 'proof finder' Agsy automates the definition of `f`: There is only one road to defining `f` so that the constraints hold by 'refl'exivity —i.e., by definition.

[3]Up to observational equality. For example, there are multiple sorting algorithms but they achieve the same end-goal.

Correct-by-Construction Programming

```
factorial :  Σ f : (ℕ → ℕ) •  f 0 ≡ 1 × (∀ {n} →  f (1 + n)  ≡  n * f n)
factorial = f , refl , refl
  where f : ℕ → ℕ
        f zero    = 1
        f (suc n) = n * f n
```

By utilising dependent types, run time errors —failures occurring during program execution, such as non-emptiness or well-formedness conditions— are transported to compile time, which are errors caught during typechecking. This is in itself a tremendously amazing feature.

Dependent types enable all errors, including logical errors, to become type checking errors!

Regarding the middle clause, *including logical errors*, suppose we are interested in a utility function whose inputs must be even numbers, or rather any computable precondition `p`. In simpler type systems, such as JavaScript's, we could throw an exception if the input does not satisfy it or simply return a `null`, which then needs to be handled at the call site by using conditionals or try-catch blocks. Instead of all of this explicit plumbing, DTLs allow us to define types and let the compiler handle the grunt work. That is, in a DTL we could encode the precondition directly into the function's type.

1.1.4 The Syntax of DTLs

Let us conclude with the minimalist's syntax[4] for dependently-typed languages.

[4]It is common to see *binder notation* as $\mathcal{Q}$ `x : τ → ⋯`, `(`$\mathcal{Q}$ `x : τ) ⋯`, and in $\mathcal{Q}$ `x : τ • ⋯`. We prefer the final form. Agda uses $\mathcal{Q}$ `(x : τ) → ⋯` for binders, and uses ∀ in-place of Π and `record`-syntax in-place of Σ. The appendix shows how to use Agda's `syntax` declarations to allow one to use nearly any desired, linear, syntax. Our demonstration language, Agda, happens to use a *hierarchy of types* which it calls `Set`$_i$ instead of `Type`$_i$; however, DTLs do not need such a hierarchy; indeed some have a single universe `Type` with the rule `Type : Type`.

DTL Syntax

```
0   -- Terms: Expressions and Types
1   e, τ ::= α            -- base type
2        | Type_i          -- “type of types”; Universe of small types at level i : ℕ
3        | ℕ               -- “Levels” for the type hierarchy
4        | Π x : τ • τ     -- ‘Pi’, dependent-function type
5        | Σ x : τ • τ     -- ‘Sigma’, dependent-sum type
6        | x               -- Variable
7        | e e             -- Application; Π-elimination
8        | λ x : τ • e     -- Abstraction; Π-introduction
9        | (e , e)         -- Pairing; Σ-introduction
10       | fst e | snd e -- Projections; Σ-elimination
11       | Fix F           -- Fixpoints for F : Type_i → Type_i
12
13  -- Abbreviation: Provided β does not refer to variable ‘_’,
14  (α → β) :=  (Π _ : α • β)
```

The Simply Typed (non-polymorphic) λ-Calculus is obtained by splitting the sole term syntactic category into two categories, the ‘types’ being the first two clauses, and the ‘expressions’ being clauses 6-8, with all other clauses dropped and taking ‘$\rightarrow$’ as a primitive type constructor. If only $\texttt{Type}_0$ and Π are then admitted with the proviso that its bound variable `x` ranges only over types —values of $\texttt{Type}_0$— then we obtain the Polymorphic λ-Calculus. Notice how the elusive notion of polymorphism is captured explicitly as ‘type abstraction’ in a DTL —with the ‘varying type argument’ becoming a legitimate and necessary argument that must be provided in function calls. For families of types[5], such as `Vec` τ `n` consisting of those lists of τ elements of length `n`, we must not only abstract over *types* τ but also over *values* `n`; hence, the *dependent function space* constructor Π generalises the usual function space ‘$\rightarrow$’. Indeed, `Vec` is only typeable in the presence of dependent-types. With intricate type class masochism Lindley and McBride [LM13], under the constraint that types are *not* terms, one is forced to duplicate term-level definitions at the type-level and thereby mimic dependent-types; albeit in a difficult fashion.

Traditionally, types constrain terms but by allowing terms to occur in types we now have *terms constraining types*; hence, there is no real distinction between the two. **Everything is a term!** In the phrase `e` : τ there is no syntactic constraint forcing `e` to be a non-type term; the symbol ‘:’ thus relates two terms. However, it is conventional to use the word *type* to refer to terms τ with τ : $\texttt{Type}_i$ for some level $i : \mathbb{N}$, and one calls $\texttt{Type}_i$ a *kind* or a *universe of (types of) level i*.

Below are the typing rules. Notice that there is only one family of rules; whereas the simply typed λ-calculus with its two syntactic categories, for expressions and terms, must have a set of rules for well-formed types then a set of rules for how to type values. Since everything is a term in a DTL, there is only one set of rules.

[5]Here are a few concrete instances: `Vec` τ `1` $\cong \tau$; `Vec` τ `2` $\cong \tau \times \tau$; `Vec` τ `3` $\cong \tau \times \tau \times \tau$.

DTL Typing Rules

```
-- Typing Rules: “Γ ⊢ e : τ” indicates term e is of type τ in (valid) context Γ
-- The context Γ documents the types of the variables that may appear in e and τ
Γ ::= ϵ | Γ, x : τ  {- Empty context; adding an identifier -}

-- Well formed contexts
(0) Valid ϵ
(1) Valid (Γ, x : τ)   ⇐   Valid Γ  ∧  Γ ⊢ τ : Typeᵢ  for some i : ℕ

-- Typing Rules
Γ ⊢ x : τ   ⇐   Γ(x) = τ  {- Γ(x) is the information assocaited with name x in Γ
 ↪  -}
Γ ⊢ α : Type₀ {- For brevity, base types are all “small types” -}
Γ ⊢ (Π x : τ • τ’) : Typeₖ   ⇐   Γ ⊢ τ : Typeᵢ
                             ∧  Γ, x : τ ⊢ τ’ : Typeⱼ for some i,j : ℕ
                             ∧  k = max{i, j}
Γ ⊢ Typeᵢ : Typeᵢ₊₁  for all i : ℕ
Γ ⊢ (λ x : τ • e) : (Π x : τ • τ’)   ⇐   Γ, x : τ ⊢ e : τ’
Γ ⊢ e e’ : τ[x ≔ e’]   ⇐   Γ ⊢ e : (Π x : τ • τ’)  ∧  Γ ⊢ e’ : τ

Γ ⊢ Fix F : Typeᵢ   ⇐   Γ ⊢ F : Typeᵢ → Typeᵢ
```

It is important to note that some type formers do not have fixed points such as **F** `X = X`. This does not matter, all we request is that *some* type is assigned to such type formers; it may not necessarily be a least fixed point but possibly, say, an empty type.

Note that Σ is included for convenience, since it could have been introduced as an abbreviation for its Church encoding; i.e., its elimination rule. That is

Σ-contexts from Π-contexts

```
(Σ x : τ • τ’) ≅  Π ρ : (Π x : τ • Π w : τ’ • Typeᵢ) • Π x : τ • Π w : τ’ • ρ x w
fst            ≅  Π e : (Σ x : τ • τ’) • e (λ x : τ • λ w : τ’ • x)
snd            ≅  Π e : (Σ x : τ • τ’) • e (λ x : τ • λ w : τ’ • w)
```

This is already illuminating for our thesis: **The record structuring mechanism Σ can be captured using the function mechanism Π.**

1.2 Burdens of DTLs: The Paralysing Paradox of Choice

Since a *dependently-typed language* is a typed language —i.e., a formal syntactic grammar and associated type system— where we can write *types* that depend on *terms*; consequently types may require non-trivial term calculation in order to be determined McKinna [McK06]. A glaring drawback is that types now depend on term calculations thereby rendering type checking, and type inference, to be difficult if not impossible Dowek [Dow93]. E.g., `Vec`

`String (factorial 100)` is the type of really long lists of strings —the length will take some time to calculate.

Unsurprisingly, "doing" dependent typing "right" is still an open issue **citet:DBLP:conf/haskell/Lindl** Brady [Bra05], Blaguszewski [Bla10], Löh, McBride, and Swierstra [LMS10], Brady [Bra], and Weirich [Wei]. In particular, after more than 30 years after Martin-Löf's work on the type theory Martin-Löf [Mar85] and Martin-Löf and Sambin [MS84], it is still unclear how such typing should be implemented so that the result is usable and well-founded. Of interest is Agda which claims to have achieved this desired ground but, in reality, it is seldom used as a programming language due to efficiency issues; in contrast, Idris aims at efficiency but its use as a proof assistant is somewhat lacking in comparison to Agda. Below are a few other issues that demonstrate the non-triviality of problems in dependently-typed languages.

1. Should programs be total for the sake of consistency or can they be partially defined?
2. Do we allow the "Type in Type" axiom Russell [Rus], Altenkirch [Alt], Cardelli [Car], and Luo [Luo90]?
3. What about "Axiom K" expressing *almost* the recursion scheme of identity types Streicher [Str93], McBride [McB00a], Cockx, Devriese, and Piessens [CDP14], Goguen, McBride, and McKinna [GMM06], McBride [McB00b], Hofmann and Streicher [HS94], and Werner [Wer08]}?
4. Should dependent pattern matching give us more information about a type? How does this interact with side effects?
5. Should unification be proof-relevant; i.e., to consider the *ways* in which terms can be made equal Cockx and Devriese [CD18]?
6. How do subtypes, which classically require proof irrelevance, tie into the paradigm?
7. How does proof-term erasure work Tejiscak and Brady [TB], Brady, McBride, and McKinna [BMM03], Mishra-Linger and Sheard [MS08], and Haselwarter [Has15]}?
8. When are two values, or programs, or types equal: When they have the same type?
9. Should a language permit non-termination or require explicit co-data?

Besides technical concerns, there are also pressing practical concerns. Since dependent types blur the distinction between value and type —thereby conflating many traditional programming concepts— library design becomes pretty delicate.

◇ For example, the method that extracts the first element of a list can in traditional languages be assigned usually two types —one with an explicit exception decoration such as Haskell's `Maybe` or C#'s `Nullable`, or without this and instead throwing an (implicit) exception. In addition, in a DTL, we can instead decorate the list with a positive length to avoid exceptions altogether, or request a non-emptiness proof, or output a dependent pair consisting of a proof that the input list is non-empty and, if so, an element of that list, or do we request as input a dependent pair consisting of a list and a non-emptiness proof —note that this is a Σ-type, in contrast to the curried form from earlier—, or $\cdots$.

◇ Moreover, when a function is written *which* properties should be attached to the resulting type and which should be stated separately?

  For example, if we write an append function for lists, do we separately prove that the length of an append is the sum of the lengths of its arguments, or do we encode that information into the return type by means of a dependent pair?

Hence programming style becomes vastly more important in DTLs since simple functions can have a diverse set of typings. In particular, this can lead to 'duplication' of code: Dependently-typed and simply typed variants of the 'same' concept, as well as the methods & proofs that operate on them; e.g., $\mathbb{N}$-indexed vectors vs. lists, Ko and Gibbons [KG13], Bernardy and Guilhem [BG13], and McBride [McB]. So much for the DRY[6] Principle. Since in a DTL records and modules are conflated, perhaps the structuring-mechanism combinators resulting from this research could reduce some of the 'duplication'.

We, as a community, are decidedly still learning about the role of dependent types in programming!

1.3 A Language Has Many Tongues

A programming language is actually many languages working together.

The most basic of imperative languages comes with a notion of 'statement' that is executed by the computer to alter 'state' and a notion of 'value' that can be assigned to memory locations. Statements may be sequenced or looped, whereas values may be added or multiplied, for example. In general, the operations on one linguistic category cannot be applied to the other. Unfortunately, a rigid separation between the two sub-languages means that binary choice, for example, conventionally invites two notations with identical semantics —e.g.; in `C` one writes `if (cond) clause`$_1$ `else clause`$_2$ for statements but must use the notation `cond ? term`$_1$ `: term`$_2$ for values. Hence, there are value and statement languages.

Let us continue using the `C` language for our examples since it is so ubiquitous and has influenced many languages. Such a choice has the benefit of referring to a concrete language, rather than speaking in vague generalities. Besides Agda —our language of choice— we

[6]Don't Repeat Yourself

shall also refer to Haskell as a representative of the functional side of programming. For example, in Haskell there is no distinction between values and statements —the latter being a particular instance of the former— and so it uses the same notation `if` $\boxed{\ldots}$ `then` $\boxed{\ldots}$ `else` $\boxed{\ldots}$ for both. However, in practice, statements in Haskell are more pragmatically used as a body of a `do` block for which the rules of conditionals and local variables change —hence, Haskell is not as uniform as it initially appears.

In `C`, one declares an integer value by `int x;` but a value of a user-defined type `T` is declared `struct T x;` since, for simplicity, one may think of `C` having an array named `struct` that contains the definitions of user-defined types `T` and the notation `struct T` acts as an array access. Since this is a clunky notation, we can provide an alias using the declaration `typedef existing-name new-name;`. Unfortunately, the existing name must necessarily be a type, such as `struct T` or `int`, and cannot be an arbitrary term. One must use `#define` to produce term aliases, which are handled by the `C` preprocessor, which also provides `#include` to 'copy-paste import' existing libraries. Hence, the type language is distinct from the libraries language, which is part of the preprocessor language.

In contrast, Haskell has a pragma language for enabling certain features of the compiler. Unlike `C`, it has an interface language using type-`class`-es which differs from its `module` language Diatchki, Jones, and Hallgren [DJH], Sheard, Harrison, and Hook [SHH01], and Sheard [She] since the former's names may be qualified by the names of the latter but not the other way around. In turn, type-`class` names may be used as constraints on types, but not so with `module` names. It may be argued that this interface language is part of the type language, but it is sufficiently different that it could be thought of as its own language Leroy [Ler00] —for example, it comes with keywords `class, instance, =>` that can only appear in special phrases. In addition, by default, variable declarations are the same for built-in and user-defined types —whereas `C` requires using `typedef` to mimic such behaviour. However, Haskell distinguishes between term and type aliases. In contrast, Agda treats aliasing as nothing more than a normal definition.

Certain application domains require high degrees of confidence in the correctness of software. Such program verification settings may thus have an additional specification language. For `C`, perhaps the most popular is the ANSI C Specification Language, ACSL Brito and Pinto [BP10]. Besides the `C` types, ACSL provides a type `integer` for specifications referring to unbounded integers as well as numerous other notions and notations not part of the `C` language. Hence, the specification language generally differs from the implementation language. In contrast, Haskell's specifications are generally Hallgren et al. [Hal+] in comments but its relative Agda allows specifications to occur at the type level.

Whether programs actually meet their specifications ultimately requires a proof language. For example, using the Frama-C tool Volkov, Mandrykin, and Efremov [VME18], ACSL specifications can be supported by Isabelle or Coq proofs. In contrast, being dependently-typed, Agda allows us to use the implementation language also as a proof language —*the only distinction is a shift in our perspective; the syntax is the same.* Tools such as Idris and Coq come with 'tactics' —algorithms which one may invoke to produce proofs— and may combine them using specific operations that only act on tactics, whence yet another tongue.

Hence, even the simplest of programming languages contain the first three of the following sub-languages —types may be treated at runtime.

1. Expression language;
2. Statement, or control flow, language;
3. Type language;
4. Specification language;
5. Proof language;
6. Module language;
7. Meta-programming languages —including Coq tactics, C preprocessor, Haskell pragmas, Template Haskell's various quotation brackets `[x| ... ]`, Idris directives, etc.

As briefly discussed, the first five languages telescope down into one uniform language within the dependently-typed language Agda. So why not the module language?

1.4 Facets of Structuring Mechanisms

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

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

1.4.1 Three Ways to Define Monoids

Recall that the signature of a monoid consists of a type `Carrier` with a method `_⨾_` that composes values and an `Id`-entity value. With Agda's lack of type-proof discrimination, i.e., its support for the Curry-Howard Correspondence, the “propositions as types” interpretation, we can encode the signature as well as the axioms of monoids to yield their theory presentation in the following two ways. Additionally, we have the derived result: `Id`-entity can be popped-in and out as desired.

The following code blocks contain essentially the same content, but presented using different notions of packaging. Even though both use the `record` keyword, the latter is treated as a typeclass since the carrier of the monoid is given ‘statically’ and instance search is used to invoke such instances.

Monoids as Agda Records

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

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

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

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

Monoids as Typeclasses

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

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

open HasMonoid {{...}} using (pop-Id-tc)
```

The double curly-braces `{{...}}` serve to indicate that the given argument is to be found by instance resolution: The derived results for `Monoid-Record` and `HasMonoid` can be invoked without having to mention a monoid on a particular carrier, provided there exists one unique record value having it as carrier —otherwise one must use named instances Kahl and Scheffczyk [KS01]. Notice that the carrier argument in the typeclasses approach, "structure on a carrier", is an (undeclared) implicit argument to the `pop-Id-tc` operation.

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

Monoids as Dependent Sums

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

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

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

Agda

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

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

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

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

There are a large number of such hyper-graphs, we have only presented a stratified view for brevity. In particular, `Monoid-Σ` is the extreme unbundled version, whereas `Monoid-Record` is the other extreme, and there is a large spectrum in between —all of which are somehow isomorphic; e.g., `Monoid-Record ≅ Σ C : Set • HasMonoid C`. Our envisioned system would be able to derive any such view at will Astesiano et al. [Ast+02] and so programs may be written according to one view, but easily repurposed for other view with little human intervention.

1.4.2 Instances and Their Use

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

Instance Declarations

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

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

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

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

No Monoids Mentioned at Use Sites

```
ℕ-pop-0ᵣ : ∀ (x y : ℕ) → x + 0 + y  ≡  x + y
ℕ-pop-0ᵣ = pop-Idᵣ

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

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

One may realise that `pop-0` proofs as a form of polymorphism —the result is independent of the particular packaging mechanism; record, typeclass, Σ, it does not matter.

Finally, let us exhibit views between the Σ form and the `record` form.

Agda

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

{- Organise a tuple componenets as implementing named fields -}
to-record-from-usual-type : Monoid-Σ → Monoid-Record
to-record-from-usual-type (c , id , op , lid , rid , assoc)
    = record { Carrier = c
             ; Id      = id
             ; _⨾_     = op
             ; lid     = lid
             ; rid     = rid
             ; assoc   = assoc
             } -- Term construed by ‘Agsy’,
               -- Agda’s mechanical proof search.
```

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

1.4.3 A Fourth Definition —Contexts

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

Monoids as Telescopes

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

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

Notice that this is nothing more than the named fields of `Monoid-Record` but not[7] bundled. Additionally, if we insert a Σ before each name we essentially regain the `Monoid-Σ` formulation. It seems contexts, at least superficially, are a nice middle ground between the previous two formulations. For instance, if we *syntactically*, visually, move the `Carrier : Set` declaration one line above, the resulting setup looks eerily similar to the typeclass formulation of records.

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

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

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

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

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

  open Monoid-Telescope-User Carrier Id _⨾_ lid rid assoc
```

Notice that we just listed the components out —rather reminiscent of the formulation `Monoid-Σ`. This observation only increases confidence in our thesis that there is no real distinctions of packaging mechanisms in DTLs.

Undeniably instantiating the telescope approach to monoids for the natural number is nothing more than listing the required components.

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

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

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

```agda
ℕ-popₘ  : ∀(x y : ℕ)  →  x + 0 + y  ≡  x + y
ℕ-popₘ  =   pop-Idₘ
```

Notice how this presentation makes it explicitly clear why we cannot have multiple instances: There would be name clashes. Even if the data we used had distinct names, the derived result may utilise data having the same name thereby admitting name clashes elsewhere. —This could be avoided in Agda by qualifying names and/or renaming.

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

Demonstrating the interdefinablity of different notions of packaging cements our thesis that it is essentially *utility* that distinguishes packages more than anything else. In particular, explicit distinctions have lead to a duplication of work where the same structure is formalised using different notions of packaging. In chapter 4 we will show how to avoid duplication by coding against a particular ‘package former’ rather than a particular variation thereof —this is akin to a type former.

1.5 Problem Statement, Objectives, and Methodology

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

1.5.1 Problem Statement

Currently, first-class module systems for dependently-typed languages are poorly *supported.* Modules $\mathcal{X}$ consisting of functions symbols, properties, and derived results are currently presented in the form Is$\mathcal{X}$: A module parameterised by function symbols and exposing derived results possibly with further, uninstantiated, proof obligations. This is understandable: Function symbols generally vary more often than proof obligations. (This is discussed in detail in Section 2.3.) However, when users do not yet have the necessary parameters they need to use a curried form of the module and so library developers also provide a module $\mathcal{X}$ which packs up the parameters as necessary fields within the module. Unfortunately, there is a whole spectrum of modules $\mathcal{X}_i$ that is missing: These are the module $\mathcal{X}$ where only `i` of the original parameters are exposed with the remaining being packed-away into the module body. It is tedious and error-prone to form all the $\mathcal{X}_i$ by hand; such 'unbundling' should be mechanically achievable from the completely bundled form $\mathcal{X}$. A similar issue happens when one wants to *describe a computation* using module $\mathcal{X}$, then its function symbols need to have associated syntactic counterparts; the tedium then increases if one considers the family $\mathcal{X}_i$.

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

1.5.2 Objectives and Methodology

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

- ◇ Objective 1: Modelling Module Systems The first objective is to actually develop a framework that models module systems —grouping mechanisms— within DTLs. The resulting framework should capture at least the expected features:

  1. Namespacing, or definitional extensions
  2. Opaque fields, or parameters
  3. Constructors, or uninterpreted identifiers

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

- ◇ Objective 2: Support Unexpected Notions of Module

  The second objective is to make the resulting framework *extensible*. Users should be able to form new exotic notions of grouping mechanisms *within* a DTL rather than 'stepping outside' of it and altering its interpreter —which may be a code implementation or an abstract rewrite-system. Ideally, users would be able to formulate arbitrary constructions from Universal Algebra and Category Theory. For example, given a theory —a notion of grouping— one would like to 'glue' two 'instances' along an 'identified common interface'. More concretely, we may want to treat some parameters as 'the same' and others as 'different' to obtain a new module that has copies of some parameters but not others. Moreover, users should be able to mechanically produces the necessary morphisms to make this construction into a pushout. Likewise, we would expect products, unions, intersections, and substructures of theories —when possible, and then to be constructed by users. In this thesis, we only want to provide a fixed set of meta-primitives from which usual and (un)conventional notions of grouping may be defined.

- ◇ Objective 3: Provide a Semantics The third objective is to provide a semantics for the resulting framework. We propose to implement the framework in the dependently-typed functional programming language Agda, thereby automatically furnishing our syntactic constructs with semantics as Agda functions and types. This has the pleasant side-effect of making the framework accessible to future researchers for experimentation.

1.6 Contributions

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

1. The ability to model module systems *for* DTLs *within* DTLs
2. The ability to arbitrarily *extend* such systems by users at a high-level
3. Demonstrate that there is an expressive yet minimal set of module meta-primitives which allow common module constructions to be defined
4. Demonstrate that relationships between modules can also be *mechanically* generated.
   - ◇ In particular, if module $\mathcal{B}$ is obtained by applying a user-defined 'variational' to module $\mathcal{A}$, then the user could also enrich the child module $\mathcal{B}$ with morphisms that describe its relationships to the parent module $\mathcal{A}$.
   - ◇ E.g., if $\mathcal{B}$ is an extension of $\mathcal{A}$, then we may have a "forgetful mapping" that drops the new components; or if $\mathcal{B}$ is a 'minimal' rendition of the theory $\mathcal{A}$, then we have a "smart constructor" that forms the rich $\mathcal{A}$ by only asking the few $\mathcal{B}$ components of the user.
5. Demonstrate that there is a *practical* implementation of such a framework
6. Solve the unbundling problem: The ability to 'unbundle' module fields as if they were parameters 'on the fly'
7. Bring algebraic data types under the umbrella of grouping mechanisms: An ADT is just a context whose symbols target the ADT 'carrier' and are not otherwise interpreted.
   - ◇ In particular, both an ADT and a record can be obtained from a *single* context declaration.

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

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

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

9. Finally, the resulting framework is *mostly type-theory agnostic*: The target setting is DTLs but we only assume the barebones as discussed in 1.1; if users drop parts of that theory, then *only* some parts of the framework will no longer apply.

   ◇ For instance, in DTLs without a fixed-point functor the framework still 'applies', but can no longer be used to provide arbitrary algebraic data types from contexts.

1.7 Related Publications

Below are works related to the research presented in this thesis.

Publication M. Al-hassy, J. Carette, and W. Kahl. A language feature to unbundle data at will (short paper). *The 18th International Conference on Generative Programming*, (Submitted 2019).

Conference M. Al-hassy, J. Carette, W. Kahl, and Y. Sharoda. Metaprogramming Agda. *The 19th IFIP Conference on Program Generation.*

Draft M. Al-hassy, J. Carette, and W. Kahl. Functional Pearl: Do-it-yourself module types. (Rejected from *The 19th International Conference on Functional Programming*)

Technical Report M. Al-hassy. Making Modules with Meta-Programmed Meta-Primitives: Liberating Package Formation from the Backend. Available: `https://alhassy.github.io/next-700-module-systems/prototype/package-former`

1.8 Structure of the Thesis

The remainder of this thesis is organised as follows.

Chapter 2 Survey existing DTL libraries to find *actual* problems with module systems that people want to solve.

Chapter 3 Survey the current state-of-the-art in literature with respect to first-class module systems in DTLs.

Chapter 4 Implementations make subtle issues less subtle, and this chapter discusses a rapid prototype for the intended framework. Besides the operational Lisp semantics, a preliminary rewrite-system semantics is given for the intended syntactic framework.

Chapter 5 With the lessons learned from the prototype, an in-language framework is developed. It is better than the prototype in some respects, but weaker in others.

Chapter 6 The contributions made by this thesis are highlighted and assessed. Conclusions are drawn and avenues for future work are suggested.

Chapter 2

Motivating the problem —Examples from the Wild

Tedium is for machines; interesting problems are for people.

In this section, we showcase a number of problems that occur in developing libraries of code, with an eye to 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 (section 2.1, 2.2) and on mixing-in features to an existing module (section 2.3, 2.4, 2.5). In order to make the core concepts acceptable, we will occasionally render examples using the simple algebraic structures: Magma , Semigroup, and Monoid [fn:4].

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, Agda's `Algebra.Morphism` "library" 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.

[WK: Chapt. 2: Agda's Algebra.Morphism library - not Agda's, but part of the Agda "standard library", or Agda std-lib - every reference to Agdaa std-lib needs to include the version referred to.

The cited sentence does not occur in stdlib-1.0.1.

Your overall description of the issue there is very fuzzy, and not understandable without looking at that module. $\boxed{]}$

2.1 Simplifying Programs by Exposing Invariants at the Type Level

In theory, lists and vectors are the same —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 by awkward or impossible in the latter case. For instance, below we demonstrate that the function `head`, which extracts the first element of a non-empty list, not only has a difficult type to read, but also requires an auxiliary relation 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.

Exposing Information At the Type Level

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

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

data not-null {A : Set} : List A → Set where
  indeed : ∀ {x xs} → not-null (x :: xs)

head : ∀ {A} → Σ xs : List A • not-null xs → A
head ([] , ())
head (x :: xs , indeed) = x

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

In the definition of `head`, 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 `not-null []`, 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 `non-null` proof. The ‘absurd pattern’ `()` indicates the impossibility of a construction and is covered later in section B.8.

This phenomenon 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. Perhaps the most popular instance of this scenario is in the setting of rings: There is an additive monoid `(R, +, 1)` and a multiplicative monoid `(R, ×, 0)` on the same underlying set `R`, and their interaction is dictated by two distributivity axioms, such as `a × (b + c) ≈ (a×b) + (a × c)`. As with `head` above, depending on which features of a monoid are exposed upfront, such axioms may be either difficult to express or relatively easy.

For brevity, since our interest is in expressing the aforementioned distributivity axiom, we shall ignore all other features of a monoid, to obtain a magma.

Distributivity is Difficult to Express

```
record Magma₀ : Set₁ where
  field
    Carrier : Set
    _⨾_      : Carrier → Carrier → Carrier

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 $\texttt{distribute}_0$. Even though the carriers of the monoids are propositionally equal, $R_+ \equiv R_\times$, they are not the same by definition — the notion of equality is defined in section B.3. As such, we are forced to "coe"rce back and forth; leaving the distributivity axiom as an exotic property of addition, multiplication, and coercions. Even worse, without the cleverness of declaring two coercion helpers, the typing of $\texttt{distribute}_0$ would have been so large and confusing that the concept would be rendered near useless.

In theory, parameterised structures are no different from their unparameterised, or "bundled", counterparts. However, in practice, this is wholly untrue: Below we can phrase the distributivity axiom nearly as it was stated informally earlier since the shared carrier is declared upfront.

Distributivity is Expressed Easily with Unbundled Structures

```
record Magma₁ (Carrier : Set) : Set₁ where
  field
    _⨾_      : Carrier → Carrier → Carrier

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

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

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

In contrast to the bundled definition of magmas, this form requires no cleverness to form coercion helpers, and is closer to the informal and usual distributivity statement.

By the same arguments above, the simple statement relating the two units of a ring `1 × 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.

Computing is filled with exciting problems; machines should help us reduce if not eliminate boring tasks.

> **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".

Observe that 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.

2.2 Renaming

The use of an idea is generally accompanied with particular notation that is accepted by the 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 people familiar with the Yoneda Lemma, or continuations, such a notation is simply "wrong"!

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

`0 × a ≈ a` The identity of "multiplicative symbols" should never resemble "0"; instead it should resemble "1" or, at least, `‘‘e’’` —the standard abbreviation of the influential algebraic works of German authors who used "Einheit" which means "identity".

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

From the few examples above, it is immediate that to even present a prototypical notation for an idea, one immediately needs auxiliary notation when specialising to a particular instance. For example, to use "additive symbols" such as `+`, $\sqcup$, $\oplus$ to denote an arbitrary binary operation leads to trouble in the function composition instance above, whereas using "multiplicative symbols" such as $\times$, $\cdot$, `*` 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,
2. The RATH-Agda library, and
3. A recent categories library.

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

1. Rename as needed.
   - ◇ There is no systematic approach to account for the many common renamings.
   - ◇ Users are encouraged to do the same, since the standard library does it this way.
2. Pack-up the *common* renamings as modules, and invoke them when needed.
   - ◇ Which renamings are provided is left at the discretion of the designer —even "expected" renamings may not be there since, say, there are too many choices or insufficient man power to produce them.

- ◇ The pattern to pack-up renamings leads nicely to consistent naming.

3. Names don't matter.
    - ◇ Users of the library need to be intimately connected with the Agda definitions and domain to use the library.
    - ◇ Consequently, there are many inconsistencies in naming.

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

2.2.1 Renaming Problems from Agda's Standard Library

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

Additive Renaming —IsNearSemiring

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

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

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

Additive Renaming Again —IsSemiringWithoutOne

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

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

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

Additive Renaming a 3rd Time and Multiplicative Renaming —IsSemiringWithoutAnnihilatingZero

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

  open IsCommutativeMonoid +-isCommutativeMonoid public
        hiding (identity^l)
        renaming ( assoc       to +-assoc
                 ; ∘-cong      to +-cong
                 ; isSemigroup to +-isSemigroup
                 ; identity    to +-identity
                 ; isMonoid    to +-isMonoid
                 ; comm        to +-comm
                 )

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

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

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

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

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

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

Orginal

```
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 ≈ _∘_
    identity^l  : 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.

2.2.2 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.

For contexts where calculation in different setoids is necessary, we provide "decorated" versions of the `Setoid'` and `SetoidCalc` interfaces:

Seotoid$\mathcal{D}$ Renamings —$\mathcal{D}$decorated Synonyms

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

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

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

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

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

SeotoidCalc𝒟 Renamings —𝒟decorated Synonyms

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

This keeps going to cover the alphabet `SetoidCalcD, SetoidCalcE, SetoidCalcF, ...,` `SetoidCalcZ` then we shift to subscripted versions `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`𝒳𝒳𝒳 takes 10 lines, for a total of at-least 600 lines!

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

2.2.3 Renaming Problems from the Agda-categories Library

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

◇ Functors have fields named `F`$_0$`,` `F`$_1$`, F-resp-≈, ...`.

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

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

◇ Such lack of concern for naming might be acceptable for well-known concepts such as functors, where some communities use `F`$_i$ to denote the object/0-cells or morphism/1-cells 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` $\Rightarrow$ `G / F` $\Leftarrow$ `G`, respectively.

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

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

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

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

Symbol Soup

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

Here are a few downsides of not renaming:

1. The type of the function is difficult to comprehend; though it need not be.
   - ◇ Take `_≈₀_ = Setoid._≈_ (F₀ Nat[Hom[C][-,c],F] (F , A))`, and
   - ◇ Take `_≈₁_ = Setoid._≈_ (F₀ F B)`,
   - ◇ Then the type says: If $\beta \approx_0 \gamma$ then `η β B ⟨$⟩ f ∘ id ≈₁ F₁ F f ⟨$⟩ (η γ A ⟨$⟩ id)` —a naturality condition!
2. The short proof is difficult to read!
   - ◇ The repeated terms such as `η β B` and `η β A` could have been renamed with mnemoic-names such as η_1, η_2 or η_s, η_t for 's'ource/1 and 't'arget/2.
   - ◇ Recall that functors `F` have projections F_i, so the "mor"phism map on a given morphism `f` becomes `F₁ F f`, as in the excerpt above; however, using RATH-Agda's naming it would have been `mor F f`.

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

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

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

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

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

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

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

Listing 1: From the Agda Standard Library on Algebra

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

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

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

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

Since the standard library uses the predicate pattern, `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 takes i-many elements —c.f., `Magma`$_0$ and `Magma`$_1$ above. However, writing these variations and functions to move between them is not only tedious but also error prone. Later on, also demonstrated in [GPCE19], we shall show how the bundled form $\mathcal{X}$ acts as *the* definition, with other forms being derived-as-needed.

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

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

2.4 Redundancy, Derived Features, and Feature Exclusion

A tenet of software development is not to over-engineer solutions; e.g., we need a notion of untyped composition, and so use `Monoid`. However, at a later stage, we may realise that units are inappropriate and so we need to drop them to obtain the weaker notion of `Semigroup` —for instance, if we wish to model finite functions as hashmaps, we need to omit the identity functions since they may have infinite domains; and we cannot simply enforce a convention, say, to treat empty hashmaps as the identities since then we would lose the empty functions. Incidentally, this example, among others, led to dropping the identity features from Categories to obtain so-called Semigroupoids.

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. 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.

For example, in the Agda-categories library one finds concepts with expressive interfaces, with redundant features, prototypically named $\mathcal{X}$, along with their minimal self-contained versions, prototypically named $\mathcal{X}$Helper. In particular, the Category type and the natural isomorphism type are instances of such a pattern. The redundant features are there to make the lives of users easier; e.g., Agda-categories states the following.

> We add a symmetric proof of associativity so that the opposite category of the opposite category is definitionally equal to the original category.

To underscore the intent, we present below a minimal setup needed to express the issue. The semigroup definition contains a redundant associativity axiom —which can be obtained from the first one by applying symmetry of equality. This is done purposefully so that the "opposite, or dual, transformer" _˘ is self-inverse on-the-nose; i.e., definitionally rather than propositionally. Definitionally equality does not need to be 'invoked', it is used silently when needed, thereby making the redundant setup worth it.

Redundancy can lead to silently used equalities

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

    -- Notice:  assocˡ ≈ sym assocʳ

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

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

> **On-the-nose Redundancy Design Pattern** [Agda-Categories]: Include redundant features if they allow certain common constructions to be definitionally equal, thereby requiring no overhead to use such an equality. Then, provide a smart constructor so users are not forced to produce the redundant features manually.

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

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

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

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

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

Simulating Default Implementations with Smart Constructors

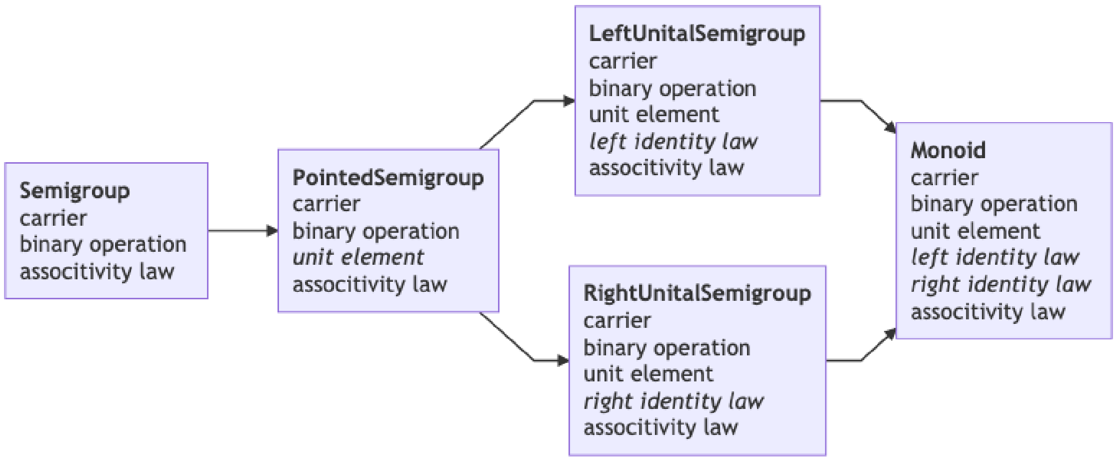
```
record Order (S : Semigroup) : Set₁ where
  open Semigroup S public
  field
    _⊑_    : Carrier → Carrier → Set
    ⊑-def  : ∀ {x y} → (x ⊑ y) ≡ (x ⨾ y ≡ x)

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

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

> **Efficient Redundancy Design Pattern** [RATH-Agda, section 17.1]: To enable efficient implementations, replace derived operators with additional fields for them and for the equalities that would otherwise be used as their definitions. Then, provide instances of these fields as derived operators, so that in the absence of more efficient implementations, these default implementations can be used with negligible penalty over a development that defines these operators as derived in the first place.

2.5 Extensions

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

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



$\boxed{\textbf{[WK:}}$ drawing: + left identity –> left-identity + right identity –> right-identity + unit element –> identity element $\boxed{]}$

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 in 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 ( ★ ) above -}
```

Extension Design Pattern: To extend a structure $\mathcal{X}$ by new features `f`$_0$`, ..., f`$_n$ which may mention features of $\mathcal{X}$, make a new structure $\mathcal{Y}$ with fields for $\mathcal{X}$, `f`$_0$`, ..., f`$_n$. Then publicly open $\mathcal{X}$ in this new structure so that the features of $\mathcal{X}$ are visible directly from $\mathcal{Y}$ to all users.

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

Extensions are not flattened inheritance

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

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

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

Extensions are not flattened inheritance

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

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, `LeftUnitalSemigroup` then all users' code would break. Instead, it would be preferable to have a 'flattened' presentation for the users that "does not leak out implementation details". We shall return to this in a future section.

It is interesting to note that diamond hierarchies cannot be trivially eliminated when providing fine-grained hierarchies. As such, we make no rash decisions regarding limiting them —and completely forgoe the unreasonable possibility of forbidding them.

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

Instead, it would be desirable to 'flatten' the hierarchy into a single package, consisting of the fields throughout the hierarchy, possibly with default implementations, yet still be able to view the resulting package at base levels in the hierarchy —c.f., section 2.4. 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.

2.6 Conclusion

After 'library spelunking', we are now in a position to summarise the problems encountered, when using existing[1] 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.

2.6.1 Lessons Learned

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

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

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

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

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

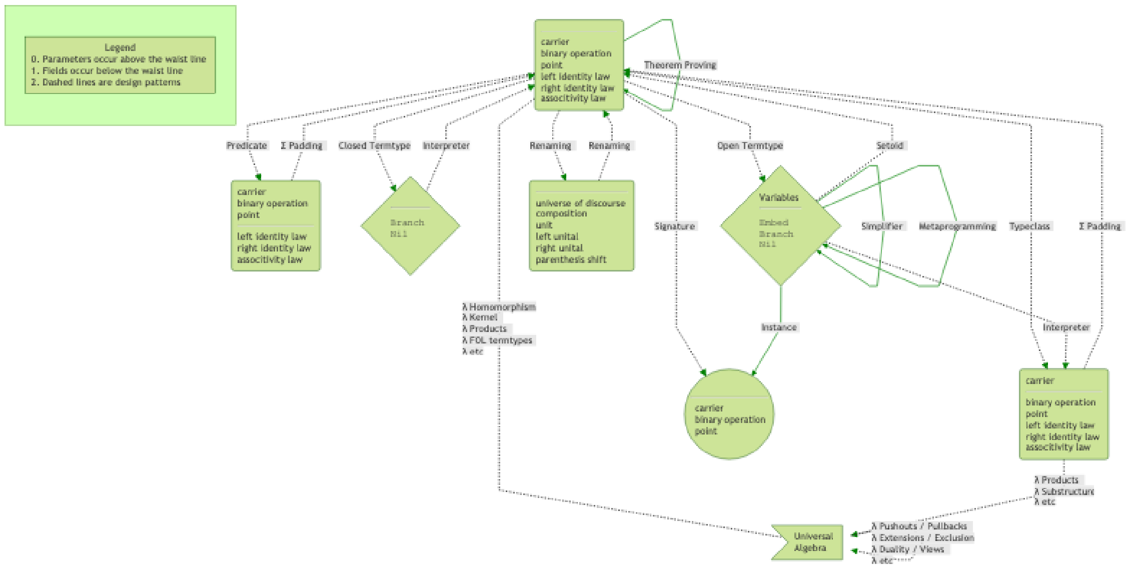


Figure 2.1: PL Research is about getting free stuff: From the left-most node, we can get a lot!

[**WK:** Fonts in drawings should not be smaller than the footnote font in the main text.]

Remarks:

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

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

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

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

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

2.6.2 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.

Chapter 3

Current Approaches

$\boxed{\textbf{[JC:}}$

- ◇ this clearly heavily borrows from your proposal (good), but it's also not clear anymore that this material 'fits' what you ended up doing. This is why having a very crisp "What problem am I solving", and then "Contributions" is so important. That will tell you what material in later sections is crucial / can be dumped.
- ◇ for example, the whole subsection on JSON feels like it brings nothing. I would delete it completely.

$\boxed{]}$. Structuring mechanisms for proof assistants are seen as tools providing administrative support for large mechanisation developments Rabe and Schürmann [RS09], with support for them usually being conservative: Support for structuring-mechanisms elaborates, or rewrites, into the language of the ambient system's logic. Conservative extensions are reasonable to avoid bootstrapping new foundations altogether but they come at the cost of limiting expressiveness to the existing foundations; thereby possibly producing awkward or unusual uses of linguistic phrases of the ambient language.

We may use the term 'module' below due to its familiarity, however some of the issues addressed also apply to other instances of grouping mechanisms —such as records, code blocks, methods, files, families of files, and namespaces.

In section 3.1 we define modularisation; in section 3.2 we discuss how to simulate it, and in section **??** we review what current systems can and cannot do; later on, in section 1.4 we provide legitimate examples of the interdefinability of different grouping mechanisms within Agda. We conclude in section 3.3 by taking a look at an implementation-agnostic representation of grouping mechanisms that is sufficiently abstract to ignore any differences between a record and an interface but is otherwise sufficiently useful to encapsulate what is expected of module systems. Moreover, besides looking at the current solutions, we also briefly discuss their shortcomings.

> The *purpose* of this section is to establish a working definition of "grouping mechanism", how it can be simulated when it is not a primitive construct, and a brief theory of their foundations which are exemplified using JavaScript.

JavaScript will be the language of choice to demonstrate these ideas since it has a primitive notion of module: Every notion of grouping mechanism boils down to begin a list of "key:value" pairs, a so-called JSON object .

3.1 Expectations of Module Systems

Packaging systems are not so esoteric that we need to dwell on their uses; yet we recall primary use cases to set the stage for the rest of our discussions.

Namespacing Modules provide new unique local scopes for identifiers thereby permitting de-coupling —possibly via multiple files contributing to the same namespace, which necessitates an independence of module names from the names of physical files; in turn, such de-conflation permits recursive modules.

Information Hiding Modules ought to provide the ability to enforce content *not* to be accessible, or alterable, from outside of the module to enforce that users cannot depend on implementation design decisions.

Citizenship Grouping mechanisms need not be treated any more special than record types. As such, one ought to be able to operate on them and manipulate them like any first-class citizen.

In particular, packages themselves have types which happen to be packages. Besides being the JavaScript approach, this is also the case with universal algebra, and OCaml, where 'structures' are typed by 'signatures'. Incidentally, OCaml and JavaScript use the same language for modules and for their *types*, whereas, for example, Haskell's recent retrofitting Kilpatrick et al. [Kil+14], of its weak module system to allow such interfacing, is not entirely in the core language since, for example, instantiating happens by the package manager rather than by a core language declaration.

Polymorphism Grouping mechanisms should group all kinds of things without prejudice.

This includes 'nested datatypes': Local types introduced for implementation purposes, where only certain functionality is exposed. E.g., in an Agda record declaration, it may be nice to declare a local type where the record fields refer to it. This approach naturally leads into hierarchical modules as well.

Interestingly, such nesting is expressible in Cayenne, a long-gone predecessor of Agda. The language lived for about 7 years and it is unclear why it is no longer maintained. Speculation would be that dependent types were poorly understood by the academics let alone the coders —moreover, it had essentially one maintainer who has since moved on to other projects.

With the metaprogramming inspired approach we are proposing, it is only reasonable that, for example, one be able to mechanically transform a package with a local type declaration into a package with the local declaration removed and a new component added to abstract it. That is, a particular implementation is no longer static, but dynamic. Real world uses cases of this idea can be found in the earlier section 2.4.

It would not be unreasonable to consider adding to this enumeration:

Sharing The computation performed for a module parameter should be shared across its constituents, rather than inefficiently being recomputed for each constituent —as is the case in the current implementation of Agda.

It is however debatable whether the following is the 'right' way to incorporate object-oriented notions of encapsulation.

Generative modules A module, rather than being pure like a function, may have some local state or initial setup that is unique to each 'instantiation' of the module —rather than purely applying a module to parameters.

SML supports such features. Whereas Haskell, for example, has its typeclass system essentially behave like an implicitly type-indexed record for the 'unnamed instance record' declarations; thereby rendering useless the interfaces supporting, say, only an integer constant.

Subtyping This gives rise to 'heterogeneous equality' where altering type annotations can suddenly make a well-typed expression ill-typed. E.g., any two record values are equal *at* the subtype of the empty record, but may be unequal at any other type annotation.

Since a package could contain anything, such as notational declarations, it is unclear how even homogeneous equality should be defined —assuming notations are not part of a package's type.

Below is a table briefly summarising the above module features for popular languages like C and JavaScript, and less popular languages Agda and OCaml.

There are many other concerns regarding packages —such as deriving excerpts, decoration with higher-order utilities, literate programming support, and matters of compilation along altered constituents— but they serve to distract from our core discussions and are thus omitted.

Concept / Language	C	JavaScript	Agda	OCaml
Namespacing	file dependent	functions and `class`	`record`	Signatures
Encapsulation	No	JSON objects	`record`	Modules
First-class modules	No	JSON objects	No	Functors
Polymorphism	Void Pointers	Dynamic	DTL	Strongly typed
Sharing	`#define`	Function args	No	Function args
Generative modules	`malloc`	Constructors, `new`	No	Yes
Subtyping	No	JSON inheritance	No	Yes

Table 3.1: How languages support module uses

3.2 Ad hoc Grouping Mechanisms

Many popular coding languages do not provide top-level modularisation mechanisms, yet users have found ways to emulate some or all of their *requirements*. We shall emphasise a record-like embedding in this section, then illustrate it in Agda in the next section. We shall number the required features then illustrate their simulation in JavaScript.

⟨0⟩ **Namespacing:** Ubiquitous languages, such as C, Shell, and JavaScript, that do not have built-in support for namespaces mimic it by a consistent naming discipline as in `theModule_theComponent`. This way, it is clear where `theComponent` comes from; namely, the 'module' `theModule` which may have its interface expressed as a C header file or as a JSON literal. This is a variation of Hungarian Notation *Hungarian notation — Wikipedia, The Free Encyclopedia* [18b].

Incidentally, a Racket source file, module, and 'language' declaration are precisely the same. Consequently, Racket modules, like OCaml's, may contain top-level effectful expressions. In a similar fashion, Python packages are directories containing an `__init__.py` file which is used for the the same purpose as Scala's `package object`'s —for package-wide definitions.

⟨1⟩ **Objects:** An object can be simulated by having a record structure contain the properties of the class which are then instantiated by record instances. Public class methods are then normal methods whose first argument is a reference to the structure that contains the properties. The relationship between an object instance and its class prototype can be viewed across a number of domains, as illustrated in the following table.

⟨2⟩ **Modules:** Languages that do not support a module may mimic it by placing "module contents" within a record. Keeping all contents within one massive record also solves the namespacing issue.

In older versions of JavaScript, for example, a module is a `json` literal —i.e., a comma separated list of key-value pairs. Moreover, encapsulation is simulated by having the module be encoded as a function that yields a record which acts as the public contents of the module, while the non-returned matter is considered private. Due to JavaScript's dynamic nature we

Template	*has a*	**Instance**
≈ class		≈ object
≈ type		≈ value
≈ theorem statement		≈ witnessing proof
≈ specification		≈ implementation
≈ interface		≈ implementation
≈ signature		≈ algebra
≈ metamodel		≈ model

Table 3.2: Muliple Forms of the Template-Instantiation Correspondence

can easily adjoin functionality to such 'modules' at any later point; however, we cannot access any private members of the module. This inflexibility of private data is both a heavy burden as well as a championed merit of the Object Oriented Paradigm.

⟨3⟩ **Sub-Modules:** If a module is encoded as a record, then a sub-module is a field in the record which itself happens to be a module encoding.

⟨4⟩ **Parameterised Modules:** If a module can be considered as encoded as the returned record from a function, then the arguments to such a function are the parameters to the module.

⟨5⟩ **Mixins:** A *Mixin is the ability to extend a datatype /X* with functionality Y long after, and far from, its definition. Mixins 'mix in' new functionality by permitting *X obtains traits Y* —unlike inheritance which declares *X is a Y*. Examples of this include Scala's traits, Java's inheritance, Haskell's typeclasses, and C#'s extension methods.

Let us see a concrete realisation of such a simulation of module features in JavaScript.

Extensions are not flattened inheritance

```
// ⟨2⟩ A simple unparamterised module with no private information
// ⟨0⟩ The field “name” is not global, but lives in a dedicated namespace
function Person (nom, age) { this.name = nom; this.age = age; }

// ⟨1⟩ An object instance;
// i.e., the dictionary literal {name: "Gödel", age: 12}
gödel = new Person("Gödel", 12)

// ⟨5⟩ Let’s mixin new functionality, say, a new method
gödel.prove = () => console.log("I have an incomplete proof...")

// ⟨2, 4⟩ A module parameterised by another module
// that is a “submodule” of “Person”.
// ⟨3⟩ The non-Person parts of the parameter are in module “P”.
function alter_module({name, age, ...P}) {

    // “Private” fields
    information = ‘I am ${name}! I am ${age} years of age!‘
    function speak() { console.log(information) }

    // The return value; fields that are promoted to “public”
    return {name, speak}
}

// Invoking the function-on-modules “alter_module”
// which mixes-in the “speak” method but drops the “age” field
kurt  = alter_module(gödel)
kurt.speak() // ⇒ I am Gödel! I am 12 years of age!

// ⟨0⟩ Notice that the “gödel” module ‘lost’ the “age” field
// when it was transformed into the “kurt” module.
console.log(kurt) // ⇒ { name: ’Gödel’, speak: [Function: speak] }
```

Typescript Bierman, Abadi, and Torgersen [BAT14] occupies an interesting position with regards to mixins: It is one of the few languages to provide union and intersection combinators for its `interface` grouping mechanism, thereby most easily supporting the Little Theories Farmer, Guttman, and Javier Thayer [FGJ92] method and making theories a true lattice. Interestingly, intersection of interfaces results in a type that contains the declarations of its arguments and if a field name has conflicting types then it is, recursively, assigned the intersection of the distinct types —the base cases of this recursive definition are primitive types, for which distinct types yield an empty intersection. In contrast, its union types are disjoint sums.

In the dependently-typed setting, one also obtains so-called 'canonical structures' Gonthier et al. [Gon+13], which not only generalise the previously mentioned mixins but also facilitate a flexible style of logic programming by having user-defined algorithms executed during unification; thereby permitting one to *omit many details Mahboubi and Tassi [MT13] and have them inferred*. As mentioned earlier regarding objects, we could simulate mixins by encoding a class as a record and a mixin as a record-consuming method. Incidentally languages admitting mixins give rise to an alternate method of module encoding: A 'module *of type M' is encoded as an instantiation of the mixin trait M.*

These natural encodings only reinforce our idea that there is no real essential difference between grouping mechanisms: Whether one uses a closure, record, or module is a matter of preference the usage of which communicates particular intent, as summarised briefly in the table below.

Concept	Possible Intent
module	Namespacing; organise related utilities under the same name
record	Bundle up related features into one 'coherent' unit
tuple	Quickly return multiple items from a function
function	An indexed value
parameterised modules	Namespaced utilities abstracted over other utilities
parameterised record	A semantic unit that 'build upon' another coherent unit

Table 3.3: Choice of grouping mechansims communicate intent

3.3 Theory Presentations: A Structuring Mechanism

Our envisioned effort would support a "write one, obtain many" approach to package formation. In order to get there, we must first understand what is currently possible. As such, we investigate how package formers are currently treated formally under the name of 'Theory Presentations'. It is the aim of this section to attest that the introduction's story is not completely on shaky foundations, thereby asserting that the aforementioned goals of the introduction are not unachievable —and the problems that posed in 2 are not trivial.

As discussed, languages are usually designed with a bit more thought given to a first-class citizen notion of grouping than is given to second-class notions of packaging-up defined content. Object-oriented languages, for example, comprise features of both views by treating classes as external structuring mechanisms even though they are normal types of the type system. This internalising of external grouping features has not received much attention with the notable mentions being Müller, Rabe, and Kohlhase [MRK18] and Dubois and Pessaux [DP15]. It is unclear whether there is any real distinction between these 'internal, integrated' and 'external, stratified' forms of grouping, besides intended use. The two approaches to Module Systems have different advantages. Both approaches permit separation of concerns: The external point of view provides a high-level structuring of a development, the internal point of view provides essentially another type which can be the subject of the language's operations —e.g., quantification or tactics— thereby being more amicable to computing transformations. Essentially it comes down to whether we want a 'module parameter' or a 'record field' —why not write it the way you like and get the other form for free.

For example, a function `f : X → Y × Z` is externally *an indexed value*, a way to structure data —`Y × Z` pairs— according to some **parameters** —`X`. By a slight change of perspective, the *type* `X → Y × Z` treated internally consists of *values* that have **field projections** $\texttt{eval}_{\times}$: For any `x : X` and `f : X → Y × Z`, we have $\texttt{eval}_{\times}$ `f : Y × Z`.

Since external grouping mechanisms tend to allow for intra-language features —e.g., imports, definitions, notation, extra-logical declarations such as pragmas— their systematic internalisation necessitates expressive record types. As such, a labelled product type or *Context* —being a list of name-type declarations with optional definitions— is a sufficiently generic rendition of what it means to group matter together.

Below is a grammar, from Müller, Rabe, and Kohlhase [MRK18], for a simple yet powerful module system based on theory (presentations) and Theory Morphisms —which are merely named contexts and named substitutions between contexts, respectively. Both may be formed modularly by using includes to copy over declarations of previously named objects. Unlike theories which may include arbitrary declarations, theory morphisms `(V : P → Q)` ≔ δ are well-defined if for every `P`-declaration `x : T`, δ contains a declaration `x ≔ t` where `t` may refer to all names declared in `Q`.

Syntax for Dependently Typed λ-calculus with Theories

```
-- Contexts
Γ  ::= ∅                                -- empty context
     | x : τ [:= τ'], Γ                  -- context with declaration, optional definition
     | Includes X, Γ                    -- theory inclusion

-- Terms
τ ::= x | τ₁ τ₂ | λ x : τ' • τ -- variables, application, lambdas
    | Π x : τ' • τ                      -- dependent product
    | [Γ] | ⟨Γ⟩ | τ.x                   -- record "[type]" and "⟨element⟩" formers,
 ↪  projections
    | Mod X                             -- contravariant "theory to record" internalisation

-- Theory, external grouping, level
θ ::= ∅                                 -- empty theory
    | X := Γ, θ                         -- a theory can contain named contexts
    | (X : (X₁ → X₂)) := Γ               -- a theory can be a first-class theory morphism

-- Proviso: In record formers, Γ must be flat; i.e., does not contain includes.
```

This concept of packaging indeed captures much of what's expected of grouping mechanisms; e.g.,

◇ Grouping mechanism should group all kinds of things and indeed there is no constraint on what a theory presentation may contain.

◇ Namespacing: Every module context can be construed as a record whose contents can then be accessed by record field projection.

  Theories as Types Müller, Rabe, and Kohlhase [MRK18] presents the first formal approach that systematically internalises theories into record types. Their central idea is to introduce a new operator `Mod` —read "models of"— that turns a theory T into a type `Mod T` which *behaves* like a record type.

◇ Operations on grouping mechanisms Carette and O'Connor [CO12].

Observe that a context is, up to syntactical differences, essentially a JavaScript object notation literal. Consequently, the notion of a mixin as described for JSON literals is here rendered as a theory morphism.

Theory Presentations	JavaScript
Context / Record	JSON object: `{key`$_0$`: value`$_0$`, ..., key`$_n$`: value`$_n$`}`
Empty context	Empty dictionary: `{}`
Inclusion	In-place syntactic unpacking: `{...`Γ`, k`$_0$`: v`$_0$`, ..., k`$_n$`: v`$_n$`}`
Theory	A file or a JSON object or an object-returning function
Translation	Function from JSON objects to JSON objects
`view`	Specification preserving translation

Table 3.4: Theory presentations in practice

For example, with the abbreviation `(Π x : A • B) = (A → B)`, we may form a small *theory* hierarchy of signatures —which is a just list of *named* contexts.

Example Theory Presentation —Informal Notation

```
  MagmaSig ≔ Carrier : Set, _⨾_ : Carrier → Carrier → Carrier, ∅
, MonSig   ≔ Includes MagmaSig, Id : Carrier, ∅
, Forget : MagmaSig → MonSig ≔ (Carrier ≔ Carrier, Id ≔ Id, ∅)
, ∅
```

This theory is then realised as follows in JavaScript —ignoring the types.

Example Theory Presentation —Executable JavaScript

```
let MagmaSig = {Carrier: undefined, op: undefined}
let MonSig   = {...MagmaSig, id: undefined}
let Forget   = (Mon) => ({Carrier: Mon.Carrier, op: Mon.op})
```

In practice, an object's features behave, to some degree, in a *known* fashion; e.g., what operators may be applied or how the object's features interact with one another. For instance, a *monoid* is an object consisting of a set `Carrier`, a value `Id` of that set, and a binary operation `_⨾_` on the set; moreover, the interaction of the latter two is specified by requesting that the operation is associative and `Id` is the identity element for the binary operation. In contrast, a *magma* is simply a set along with a binary operation. As such, the translation `Forget`, above, not only gives us a translation of features, but it also satisfies all zero coherence laws of a magma.

As mentioned earlier, a theory morphism, also known as a *view* , or Substitution, is a map between contexts that implements the interface of the source using utilities of the target; whence results about specific structures can be constructed by transport along views Farmer, Guttman, and Javier Thayer [FGJ92]: A view $\mathtt{V} : \mathcal{S} \to \mathcal{T}$ gives rise to a term homomorphism $\mathcal{V}$ from `P`-terms to `Q`-terms that is type-preserving in that whenever $\theta,\ \mathcal{S} \vdash \mathtt{e} : \tau$ then $\theta,\ \mathcal{T} \vdash \mathcal{V}\ \mathtt{e} :\ \mathcal{V}\ \tau$. Thus, views preserve judgements and, via the propositions-as-types representations, also preserve truth.

More concretely, a view $\mathtt{V} = (\mathtt{U},\ \beta) : \mathcal{S} \to \mathcal{T}$ is essentially a predicate U, of the target theory, denoting a *universe of discourse* along with an arity-preserving mapping β of $\mathcal{S}$-symbols, or declarations, to $\mathcal{T}$-expressions —by itself, β is called a *translation* . It is lifted to terms as follows —notice that the translated variable-binders are relativised to the new domain.

$\mathcal{V}$ Extended to Terms

$\mathcal{V}\ \mathtt{x} \approx \mathtt{x}$	If `x` is an $\mathcal{S}$-variable symbol
$\mathcal{V}(\mathtt{f}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n) \approx (\beta\ \mathtt{f})\ (\mathcal{V}\ \mathtt{e}_1)\ \ldots\ (\mathcal{V}\ \mathtt{e}_n)$	If `f` is an `n`-ary $\mathcal{S}$-function symbol
$\mathcal{V}(\mathcal{Q}\ \mathtt{x} \bullet \mathtt{P}) \approx (\mathcal{Q}\ \mathtt{x} \mid \mathtt{U}\ \mathtt{x} \bullet \mathcal{V}\ \mathtt{P})$	If $\mathcal{Q}$ is a variable-binder $\forall,\ \exists,\ \lambda$

The *Standard Interpretation Theorem* Farmer [Far93] provides sufficient conditions for a translation to be an 'Interpretation' which transports results between formalisations. It states: A translation is an interpretation provided $\mathcal{S}$-axioms `P` are lifted to theorems $\mathcal{V}\ \mathtt{P}$, the universe of discourse is non-empty $\exists\ \mathtt{x} \bullet \mathtt{U}\ \mathtt{x}$, and the interpretation of the universe contains the interpretations of the symbols; i.e., for each $\mathcal{S}$-symbol `f` of arity `n`,
$\mathcal{V}(\forall\ \mathtt{x}_1,\ \ldots,\ \mathtt{x}_n \bullet \exists\ \mathtt{y} \bullet \mathtt{f}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n = \mathtt{y})$ holds.

By virtue of being a validity preserving homomorphism, a standard interpretation syntactically and semantically embeds its source theory in its target theory. The most important consequence of interpretability is the *Standard Relative Satisfiability* Farmer [Far93] which says that a theory which is interpretable in a satisfiable theory is itself satisfiable; in programming terms this amount to: *If X is an implementation of* ***interface*** *$\mathcal{T}$ and $\mathcal{S}$ is interpretable in $\mathcal{T}$ then X can be transformed into an implementation of $\mathcal{S}$.* Interestingly such 'subtyping' can be derived in a mechanical fashion, but it can force the subtype relation to be cyclic. However, it is unclear under which conditions translations automatically give rise to interpretations: Can the issue be relegated to syntactic manipulation only?

Theory interpretation has been studied for first-order predicate logic then extended to higher-order logic Farmer [Far93]. The advent of dependent-types, in particular the blurring of operations and formulae *Curry–Howard correspondence — Wikipedia, The Free Encyclopedia* [18a], means that propositions of a language can be encoded into it as other sorts, dependent on existing sorts, thereby questioning *what it means to have a validity-preserving morphism* when the axioms can be encoded as operations? As far as we can tell, it seems very little work regarding theory interpretations has been conducted in dependently-typed settings Palmgren and Stoltenberg-Hansen [PS90], Baillot and Lago [BL16], Fiadeiro and Maibaum [FM93], and Lipton [Lip92].

In subsequent sections, **??** and 4.3.1, we shall identify a number of views that are formed *syntactically* and the fact that they are indeed views then becomes the need to mechanically provide certain values —which by the propositions-as-types view means we mechanically provide certain "proofs of propositions". Incidentally, moving forward, we shall consider an essentially untyped setting in which to perform such syntax shuffling —that is, even though we are tackling DTLs, we shall follow a JavaScript-like approach with essentially *one* notion of grouping rather than a theory presentation approach with two notions.

3.4 "JSON is Foundational": From Prototypes to Classes

In the previous section, we indicated that going forward, we will be taking a JSON-like approach to working with modules. JavaScript has the reputation of being non-academic, along with its dynamically type-checked nature it is not surprising that the reader may take pause to consider whether our inclination is, plainly put, 'wrong'. To reassure the reader, we will show how JSON objects are a foundational way to group data by deriving the notion of a `class` from object-oriented programming. In fact, recent implementations of JavaScript have a `class` keyword which, for the most part, is syntactic sugar for JSON objects.

We shall arrive at the `class` keyword as a means of moving away from design patterns and going to mechanical constructs.

3.4.1 Prototypical Concepts

In English, *prototype* means a preliminary model of something from which other forms are developed or *copied*. As such, a *prototypical* object is an object denoting the original or typical form of something.

In addition to their properties, JavaScript objects also have a prototype —i.e., another object that is used as a source of additional properties. When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on.

<div align="center">A *prototype*</div>

is another object that is used as a fallback source of properties. |

Adding new features or overriding methods are another primary use for prototypes. E.g., to attach a new property to a 'kind' of object, we simply need to attach it to the prototype —since all those 'kinds' of objects use the prototype's properties. In this way, we overload a method by attaching it to prototypes. If, instead, we add the property to an object, rather than to its prototype, then the property is attached directly to the object and possibly shadowing the property of the same name that the prototype has, whence overriding.

1. Prototype Example

   Prototypes let us define properties that are the same for all instances, but properties that differ per instance are stored directly in the objects themselves. E.g., the prototypical person acts as a container for the properties that are shared by all people. An individual person object, like `kathy` below, contains properties that apply only to itself, such as its name, and derives shared properties from its prototype.

   Painfully Initialising the Infrastructure of an Instance

   ```
   // An example object prototype
   let prototypicalPerson    = {};
   prototypicalPerson._world = 0;
   prototypicalPerson.speak  = function () {
     console.log(`I am ${this.name}, a ${this.job}, in a world of `
                 + `${prototypicalPerson._world} people.`) }
   prototypicalPerson.job = `farmer`;

   // Example use: Manually ensure the necessary properties are setup
   // and then manually increment the number of people in the world.
   let person = Object.create(prototypicalPerson);
   person.name = `jasim`;
   prototypicalPerson._world++;
   person.speak() // ⇒ I am jasim, a farmer, in a world of 1 people.

   // Another person requires just as much setup
   let kathy = { ...prototypicalPerson }; // Same as "Object.create(···)"
   kathy.name = `kathy`;
   prototypicalPerson._world++;
   kathy.speak() // ⇒ I am kathy, a farmer, in a world of 2 people.
   ```

   You can use `Object.create` to create an object with a specific prototype. The default prototype is `Object.prototype`. For the most part, `Object.create(someObject)` ≈ `{ ...someObject }`; i.e., we *copy* the properties of `someObject` into an empty object, thereby treating `someObject` as a prototype from which we will build more sophisticated objects.

Notice that we have to manually update the 'class variable' `_world` each time a new person instance is created.

2. Manual Constructor Functions

Classes are prototypes along with constructor functions!

A *class* defines the shape of a kind of object; i.e., what properties it has; e.g., a Person can `speak`, as all people can, but should have its own `name` property to speak of. This idea is realised as a prototype along with a *constructor* function that ensures an instance object not only derives from the proper prototype but also ensures it, itself, has the properties that instances of the class are supposed to have.

Using a Function to Initialise the Infrastructure of an Instance

```
let prototypicalPerson    = {};
prototypicalPerson._world = 0;
prototypicalPerson.speak  = function () {
  console.log(`I am ${this.name}, a ${this.job}, in a world of `
              + `${prototypicalPerson._world} people.`) }

function makePerson(name, job = `farmer`) {
  let person  = Object.create(prototypicalPerson);
  person.name = name;
  person.job  = job;
  prototypicalPerson._world++;
  return person;
}

// Example use
let jasim = makePerson(`jasim`);
jasim.speak() // ⇒  I am jasim, a farmer, in a world of 1 people.

makePerson(`kathy`).speak()
// ⇒  I am kathy, a farmer, in a world of 2 people.
```

Notice that we did not have to manually update the `_world` variable each time a new person instance is created.

3. Constructor Functions with `new` We can fuse the previous two approaches under one name by making the prototype a part of the constructor.

Constructor Functions

```
function Person(name, job = ‘farmer‘) {
 this.name = name;
 this.job  = job;
 Person.prototype._world++;
}

Person.prototype._world = 0;
Person.prototype.speak = function () {
  console.log(‘I am ${this.name}, a ${this.job}, in a world of ‘
               + ‘${Person.prototype._world} people.‘) }

// Example use
let jasim = Object.create(Person.prototype)
Person.call(jasim, ‘jasim‘)
jasim.speak() // ⇒  I am jasim, a farmer, in a world of 1 people.

// Example using shorthand
let kasim = new Person (‘kathy‘)
kasim.speak()  // ⇒  I am kathy, a farmer, in a world of 2 people.
```

If you put the keyword `new` in front of a function call, the function is treated as a constructor. This means that an object with the right prototype is automatically created, bound to `this` in the function, and returned at the end of the function.

Definition of ‘new’

```
  new f(args)
≈ (_ => let THIS = Object.create(f.prototype);
        f.call(THIS, args); return THIS;) ()
```

All functions automatically get a property named `prototype`, which by default holds a plain, empty object that derives from `Object.prototype`. You can overwrite it with a new object if you want. Or you can add properties to the existing object, as the example does.

Notice that the `Person` object *derives* from `Function.prototype`, but also has a *property* named `prototype` which is used for instances created through it.

Sanity Checks

```
console.log( Object.getPrototypeOf(Person) == Function.prototype
           , Person instanceof Function
           , jasim  instanceof Person
           , Object.getPrototypeOf(jasim) == Person.prototype)
```

Hence, we can update our motto:

Classes are constructor functions with a prototype property!

4. `class` Notation Rather than declaring a constructor, *then* attaching properties to its prototype, we may perform both steps together using `class` notation shorthand.

   Classes as Syntactic Convenience

   ```
   class Person {
     static #world = 0
     constructor(name, job = 'farmer') {
       this.name = name;
       this.job  = job;
       Person.#world++;
     }
     speak() {
       console.log('I am ${this.name}, a ${this.job}, in a world of '
                  + '${Person.#world} people.')
     }
   }

   // Example use

   let jasim = new Person('jasim')
   jasim.speak()
   // ⇒ I am jasim, a farmer, in a world of 1 people.

   new Person('kathy').speak()
   // ⇒ I am kathy, a farmer, in a world of 2 people.
   ```

   Notice that there is a special function named `constructor` which is bound to the class name, `Person`, outside the class. The remainder of the class declarations are bound to the constructor's prototype. Thus, the earlier class declaration is equivalent to the constructor definition from the previous section. It just looks nicer.

   ◇ Actually, this is even better: The `static #world = 0` declaration makes the property `world` *private*, completely inaccessible from the outside the class. The `static` keyword attaches the name not to particular instances (`this`) but rather to the constructor/class name (`Person`).

   ◇ Indeed, in the previous examples we could have accidentally messed-up our world count. Now, we get an error if we write `Person.#world` outside of the class.

3.4.2 Conclusion

Historically, physicists believed that matter was built from indivisible building blocks called *atoms*, then some hundred years later it was discovered that atoms are in-fact not atomic but are built from *neutrons, protons*, and *electrons*, then some fifty years later it was discovered that neutrons and protons are built from so called *quarks*. Similarly, albeit ironically, early versions of JavaScript were considered incomplete from an object-oriented perspective since they did not have a primitive, atomic, `class` construct. Akin to physicists, we have seen how JavaScript indeed has classes and is thus a full-fledged object-oriented language, only unlike other languages, they are not a primitive but a derived construct.

Unsurprisingly, other features of object-oriented programming can also be derived —and possibly more flexibly than their counterparts in languages that take them as primitive. For example, it can be useful to know whether an object x was derived from a specific class y and so there is the abbreviation:
x `instanceof` $y \approx$ `Object.getPrototypeOf(`x`) ==` y`.prototype`. Inheritance is then an abbreviation for using the previously discussed `Object.create(parentPrototype)` method. Finally, It can be pragmatic to have a few technical methods show up in all objects, such as `toString`, which converts an object to a string representation. To accomplish this, JavaScript's *standard library* objects have `Object.prototype` as their great ancestral prototype. In languages were classes are primitive, `Object` is the top of the class hierarchy.

Maximal Elements in the Class Hierarchy

```
// “Object” is maximal
console.log(Object.getPrototypeOf(Object.prototype)); // ⇒ null

// Empty object that *does* derive from “Object”
let basic = {}
console.log( basic instanceof Object // ⇒ true
           , "toString" in basic)    // ⇒ true

// Empty object that does not derive from “Object”
let maximal = Object.create(null);
console.log( maximal instanceof Object // ⇒ false
           , "toString" in maximal)    // ⇒ false
```

However, since JavaScript's classes are a derived concept, `Object` is not the *maximum* class but rather a *maximal* class: It has no parent class, but is not necessarily the parent of all other classes. Indeed, a declaration `let basic = {}`, by default, creates an empty object whose parent is `Object` —so as to have the aforementioned useful technical methods. If you pass `null` to `Object.create`, as shown above, the resulting object will not derive from `Object`. This is exhilarating.

So objects do more than just hold their own properties. They have prototypes, which are other objects. They'll act as if they have properties they don't have as long as their prototype has that property.

Chapter 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. As such, we begin by forming an 'editor extension' to Agda with an eye toward the minimal number of 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 2.

For the interested reader, the full implementation is presented literately as a discussion at the following website. We will not be discussing any Lisp code in particular.

`https://alhassy.github.io/next-700-module-systems/prototype/package-former.html`

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 checked language.

A lack of static types means some design decisions can be deferred as much as possible.

Why an editor extension? Metaprogramming is notoriously difficult to work with in typed settings, which mostly provide an opaque `Term` type thereby essentially resolving to

[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.

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.

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, begin 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`.

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 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` tool can generate the many common design patterns discussed earlier in Section 2.6.1; 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.

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 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, but we use the name `PackageFormer` instead of context, signature, telescope, nor theory since those names have existing biased connotations —besides, the new name is more 'programmer friendly'.

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

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

Different Ways to Organise M-Sets

```
9   Semantics    = M-Set ⊕⟶ record
10  Semantics𝒟   = Semantics ⊕⟶ rename (λ x → (concat x "𝒟"))
11  Semantics3   =  Semantics :waist 3
12
13  Left-M-Set  = M-Set ⊕⟶ record
14  Right-M-Set = Left-M-Set ⊕⟶ flipping "_·_" :renaming "leftId to rightId"
15
16  ScalarSyntax = M-Set ⊕⟶ primed ⊕⟶ data "Scalar'"
17  Signature    = M-Set ⊕⟶ record ⊕⟶ signature
18  Sorts        = M-Set ⊕⟶ record ⊕⟶ sorts
19
20  𝒱-one-carrier   = renaming "Scalar to Carrier; Vector to Carrier"
21  𝒱-compositional = renaming "_×_ to _⨾_; _·_ to _⨾_"
22  𝒱-monoidal      = one-carrier ⊕⟶ compositional ⊕⟶ record
23  LeftUnitalSemigroup = M-Set ⊕⟶ monoidal
24  Semigroup           = M-Set ⊕⟶ keeping "assoc" ⊕⟶ monoidal
25  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. This 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 `⊕⟶` $\cdots$ `⊕⟶` v_n is tantamount to explicit forward function application v_n `(`v_{n-1} `(`$\cdots$ `(`v_1 `p)))`. With this understanding, we can explain the different ways to organise M-sets.

Line 1 The context of M-sets is declared.

This is the traditional Agda syntax `record M-Set : Set`$_1$ `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 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`.

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
{- 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 13, mentions another combinator `_flipping_ : PackageFormer → Name → 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 3.3— 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.

Line 16 An algebraic data type is a tagged union of symbols, terms, and so is one type. 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.

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 20 Declarations starting with "$\mathcal{V}$-" indicate that a new variation is to be formed, rather than a new grouping mechanism.

The user-defined `one-carrier` variational identifies both the `Scalar` and `Vector` sorts, whereas `compositional` identifies the binary operations; `monoidal` then 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 mentioned, 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 Agsy proof finder.

4.3 Practicality

Herein we demonstrate how to use this system from the perspective of *library designers*. We use constructs that are discussed in the next section —which are examples of how users may extend the system to produce grouping mechanisms for any desired purpose. The exposition here follows section 2 of the *Theory Presentation Combinators* Carette and O'Connor [CO12], reiterating many the ideas therein.

The few constructs demonstrated in this section not only create new grouping mechanisms from old ones, but also create maps from the new, child, presentations to the old parent presentations. Maps between grouping mechanisms are sometimes called *views*, Section 3.3. 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. How? You, 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.

This section demonstrates the power and expressivity of the meta-primitives by showcasing a series of ubiquitous combinators *which may be defined using the meta-primitives and Lisp*. In particular, **this section showcases a core kernel of context combinators** and the section afterwards goes into the detail of how to **extend the system to build —presumably— any desired operations on any notion of grouping mechanism**.

4.3.1 Extension

The simplest situation is where the presentation of one theory is included, verbatim, in another. Concretely, consider `Monoid` and `CommutativeMonoid`.

Manually Repeating the entirety of 'Monoid' within 'CommutativeMonoid$_0$'

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

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

As expected, the only difference is that `CommutativeMonoid`$_0$ adds a `comm`-utative 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"
-}
```

Hovering over the left-hand-side gives a tooltip showing the resulting elaboration, which is identical to `CommutativeMonoid`$_0$ along with a forgetful operation ˆ_ˆ The tooltip shows the *expanded* version of the theory, which is **what we want to specify but not what we want to enter manually**. 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.

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 `▯-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`.

Anyhow, notice that we may define `GroupR` —a record-presentation of groups— as an extension of `MonoidR` using a *single* `extended-by` clause where the necessary items are separated by `;`.

Groups are monoids extended by a few new features

```
{-700
GroupR = MonoidR extended-by "_⁻¹ : Carrier → Carrier; left⁻¹ : ∀ {x} → (x ⁻¹) ·
 ↪   x ≡ ▯; right⁻¹ : ∀ {x} → x · (x ⁻¹) ≡ ▯" ⊕⟶ record
-}
```

A more fine grained approach may be as follows.

A fine-grained hierarhcy under groups

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

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[3].

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

The ‘postulating’ variational

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

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

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

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

   With this variational, a definition is only written once.
   "
   extended-by (s-replace "op" bop (s-replace "rel" using (s-replace "op’" using
    (pcase prop
     ("associative"   "assoc : ∀ x y z → op (op x y) z ≡ op x (op y z)")
     ("commutative"   "comm  : ∀ x y   → op x y ≡ op y x")
     ("idempotent"    "idemp : ∀ x     → op x x ≡ x")
     ("involutive"    "inv   : ∀ x     → op (op x) ≡ x") ;; assuming bop is unary
     ("left-unit"     "unit^l : ∀ x y z → op e x ≡ e")
     ("right-unit"    "unit^r : ∀ x y z → op x e ≡ e")
     ("distributive^l" "dist^l : ∀ x y z → op x (op’ y z) ≡ op’ (op x y) (op x z)")
     ("distributive^r" "dist^r : ∀ x y z → op (op’ y z) x ≡ op’ (op y x) (op z x)")
     ("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’)
     ;; (_ (error "𝒱-postulating does not know the property “%s”" prop))
     )))) :adjoin-retract ’adjoin-retract)
```

(The syntax of declaration is discussed in section 4.4.1.)

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 idea that *"each piece of mathematical knowledge should be formalised only once"* is prompted in [GS10].

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`.

Since renaming and extension (including postulating) both adjoin retract morphisms, by default, we are lead to wonder how 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 the postulating, we have that the operations commute if *only* the first permits the creation of a retract. Here's a concrete example:

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

These both elaborate to the same thing, up to order of constituents.

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. You, as the user, can make these alternative combinators as follows:

```
{-700

𝒱-renaming' by = renaming 'by :adjoin-retract nil

𝒱-postulating' p bop (using) = postulating 'p 'bop :using 'using :adjoin-retract
 ↪  nil

-- Example use: We need the "𝒱-" in the declaration site, but not in use sites, as
 ↪  below.

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

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.

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 _·_"
-}
```

Alternatively, renaming 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.

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

```
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}
```

We are using gensym’s for λ-arguments to avoid name clashes.

4.3.4 Union (and intersection)

But even with these features, given `GroupR`, 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; `CommutativeGroupR`$_0$ then closes this diamond-loop by adding both features. The simplest way to share structure is to union two presentations:

```
{-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 $f : X \to A$ and $g : X \to B$ is, essentially, the disjoint sum of A and B where embedded elements are considered 'indistinguishable' when the share the same origin in X via the paths f and g. Unfortunately, the resulting 'indistinguishable' elements are actually distinguishable: They may be the A-name or the B-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.

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 X 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 X to names of A and B. If we instead consider $f' : X' \leftarrow A$ and $g' : X' \leftarrow B$, in the *opposite direction*, then we may reconstruct a pushout by setting X to be common image of f', g', and set f, g to be inclusions In-particular, the full identity of X' is not necessarily relevant for the pushout reconstruction and so it may be omitted. Moreover, the issue of canonical names is resolved: If $a \in A$ is intended to be identified with $b \in B$ such that the resulting element has c as the chosen canonical name, then we simply require $f' \, a = c = g' \, b$.

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 $f : X \to A$ and $g : X \to B$ is intended to be the 'smallest object P that contains a copy of A and of B sharing the common substructure X', and as such it outputs two functions $inj_1 : A \to P$, $inj_2 : B \to P$ that inject the names of A and B into P. If we realise P as a record —a type of models— then the embedding functions are *reversed*, to obtain projections $P \to A$ and $P \to B$: If we have a model of P, then we can forget some structure and rename via f and g to obtain models of A and B. For the resulting construction to be useful, these names could be automated such as $toA : P \to A$ and $toB : P \to B$ 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 X 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.

Pushout combinator with 6 optional arguments

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


 = "Union parent PackageFormer with given PF.

   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-RETRACT_i, for i : 1..2, are the optional names of the resulting morphisms.
   Provide nil if you do not want the morphisms adjoined.

   ERROR-ON-NAME-CLASHES toggles whether the program should crash if the PackageForme
   have items with the same name but different types or definitions,
   or otherwise it should simply, and sliently, rename the conflicting names accordi
   a function that takes 3 string arguments and yields two, the former being the name
   along with the conflicting name, and yiedling two new names.

   Also, ERROR-ON-NAME-CLASHES toggles whether the program should crash if retract
   names already exist, or otherwise it should simply silently not include clashing r
   "
  :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 n
           (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))
           )

      ;; Ensure no name clashes!
      (if error-on-name-clashes
          (if name-clashes
            (-let [debug-on-error nil]
              (error "%s = %s union %s \n\n\t\t → Error: Elements ‘%s’ conflict!\n\n\
                     $name $parent p (element-name (caar name-clashes)) (s-join "\n\t\
        ;; Else handle clashes
        (loop for n in (mapcar #'element-name (apply #'-concat name-clashes))
              do (setq es₁ (--map (map-name (λ m → (if (equal n m) (car (fix-conflict
                 (setq es₂ (--map (map-name (λ m → (if (equal n m) (cdr (fix-conflict
        (setq es' (-concat es₁ es₂)))
```

The reader is not meant to understand the definition provided here, however we 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.

Here are some examples of this construction of mine.

Here we provide all arguments, optional and otherwise.

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

```
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 = _×_}
```

Remember, *this particular user implementation* realises `X`$_1$ `union X`$_2$ `:renaming`$_1$ `f' :renaming`$_2$ `g'` as the pushout of the inclusions `f' X`$_1$ $\cap$ `g' X`$_2$ $\longrightarrow$ `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.

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}
```

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
-}
```

```
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’}
```

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.

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

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

Let’s close with the classic example of forming a ring structure by combining two monoidal structures. This example also serves to further showcasing how using $\mathcal{V}$`-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
-}
```

```
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)
```

Following the reasoning for pushouts, we implement pullbacks in the same way with the same optional arguments. Here's an example use:

```
{-700
Just-Carrier     = Additive intersect Multiplicative
Magma-yet-again = Additive intersect Multiplicative :renaming₁ "_+_ to op"
 ↪   :renaming₂ "_×_ to op"
-}
```

Moreover the absorptive law $X \cap (X \cup Z) = X$ also holds for these operations: `Additive intersect AddMult` is just `Additive`, when we ignore all adjoined retracts.

4.3.5 Duality

Maps between grouping mechanisms are sometimes called *views*, which are essentially an internalisation of of the *variationals* in our system. Let's demonstrate an example of how dual concepts are captured concretely in the system.

For example, the dual, or opposite, of a binary operation `_·_` is the operation `_·`^{op}`_` defined by `x ·`^{op} `y ≔ y · x`. Classically in Agda, duality is utilised as follows:

1. Define a module `R _·_` for the desired concepts.
2. Define a shallow module `R`^{op} `_·_` that opens `R _·`^{op}`_` and renames the concepts in `R` by the dual names.

   The RATH-Agda library performs essentially this approach, for example for obtaining `UpperBounds` from `LowerBounds` in the context of a poset.

Unfortunately, this means that any record definitions in `R` must have its field names be sufficiently generic to play *both* roles as 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 lub/glb 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.

Consider the following heterogeneous algebra.

```
{-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:

```
{-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 `to⋯` 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:

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

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

  _ : let _⊒_ = λ x y → y ⊑ x
      in ∀ {x y z} → x ⊒ y → y ⊒ z → x ⊒ z
  _ = 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]][mathscheme] approach to library design: New structures are built from old ones by augmenting one concept at a time, then one uses mixins such as `union`, below, 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 [design-patterns-solid]: /No client should be forced to depend on methods it does not use.*

```
{-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"
-}
```

The cool thing here is that `CommutativeMagma` comes with `toMagma, toType,` and `toEmpty`.

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 an '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`, from earlier, but later wished to utilise other substrata. This is easily achieved with the following declarations.

```
{-700
Empty’’        = Monoid keeping ""
Type’’         = Monoid keeping "Carrier"
Magma’’        = Monoid keeping "_·_"
Semigroup’’    = Monoid keeping "assoc"
PointedMagma’’ = Monoid 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 more expressive than necessary. Such an observation no longer need to be found by inspection, instead it may be derived mechanically.

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

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

```
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 TODO 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.

The following listing of structures was 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 the web lists of Peter Jipsen and 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.

200+ One Line Specfications

~1500 Lines of Typechecked Agda

If anything, this elaboration demonstrates our tool as 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 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 wrt to the given relation; 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.

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

We implore the readers to build upon our code of theories above by, for example, define the notion of homomorphism for every single one of the theories. Besides being tiresome, such a manual process is also error-prone. Instead, one can automatically derive this concept!

Likewise, for other concepts from universal algebra —which is useful to computer science in the setting of specifications.

4.4 Semantics

Herein we demonstrate how with a little bit of Lisp[4], one may create any desired form of grouping mechanism as well as operation between groupings.

Rather than present the implementation, we shall present an abstract interpreter —-a relation '$\rightsquigarrow$' that specifies how terms 'reduce'. To present the rules for this relation, we will use an abbreviated form of contexts —which is not valid concrete syntax.

Linear Abbreviation for PackageFormer Contexts

```
  Name  = ⟨k; ℓ; qᵢ ηᵢ : τᵢ ≔ δᵢ⟩ᵢ
≈
  k Name : Set ℓ where

     q₀ η₀ : τ₀
        η₀ = δ₀

     ⋮

     qₖ ηₖ : τₖ
        ηₖ = δₖ
```

A `PackageFormer` context is simply two tags, a 'kind' `k` and a level ℓ, along with a list of 'elements' which consist of components `qualifier` $\mathtt{q}_i$, `name` η_i, `type` τ_i, `equations` definitions δ_i—the first and last are optional.

4.4.1 Declaration Rules

Begin extensible, the system allows user definable variationals which can then be applied create new contexts. For instance, the simplest user definable variational, the empty one, could be defined and used as follows.

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

User-defined variational and application thereof

```
{-700
-- Variational with empty right hand side.
𝒱-identity =

-- Using it to form a new context
MonoidP^id = MonoidP identity
-}
```

The prefix 𝒱- signals to the Elisp meta-program that this particular equation is intended to be a variational and should be *loaded into Emacs* as such. Indeed, you may view the documentation and *elaborated* Lisp of this definition using `C-h o RET 𝒱-identity`.

The prefix 𝒱- only occurs at the definition site, the call site omits it. Why? We have augmented the Emacs system with a new functional definition, and the 𝒱- serves as a namespace delimiter.

Loading the meta-program using Agda's usual `C-c C-l` lets us hover over `MonoidP`^{id} to see its elaboration is precisely that of `MonoidP`.

Moreover, to be useful, all variationals have tooltips showing their user-defined documentation. If we hover over `identity`, we are informed that it is undocumented. User documentation is optional and may appear immediately following the `=`, as follows.

Documented User-defined Variational

```
{-700
𝒱-Id = "This is the do-nothing variational"
-}
```

Operationally, we substitute equals-for-equals.

No Variational Clauses Needed

```
{-700
-- No variational clauses needed
MonoidP^0  = MonoidP
-}
```

We may also augment a variational with positional and (optional) keyword arguments that have default values. The keyword arguments along with their default value, *if any*, are enclosed in parenthesis.

User-defined Variational with Arguments

```
{-700
𝒱-test positional (keyword 3) another = "I have two mandatory arguments and one
 ↪  keyword argument"

Monoid-test = MonoidP ⊕⟶ test "positional arg₁" "positional arg₂" :keyword 25
-}
```

We are not doing anything with the arguments here; we shall return to this in later subsections.

In summary, declarations provide an alias and one may substitute equals for equals; however, only variational declarations support arguments.

$$\frac{\texttt{l = r}\text{ is declared}}{l \rightsquigarrow r}$$

$$\frac{\mathcal{V}\texttt{-l a = r}\text{ is declared}}{p \oplus\!\!\rightarrow\ l\, e \rightsquigarrow p \oplus\!\!\rightarrow\ r[a \coloneqq e]}$$

Ideally variational definition would be rendered in Agda code; we will return to this issue in Section 5.

Declaration Well-definedness Provisos: A declaration `l = r` must satisfy:

1. The name `l` is a string of consecutive symbols, if this is a context declaration; otherwise, `l` must be of the form $\mathcal{V}$`-ll` $\mathtt{a}_0$ ... $\mathtt{a}_n$ to designate it as a variational declaration with arguments $\mathtt{a}_i$ which in turn are either atomic names or pairs `(n d)` consisting of an atomic name along with a default value.
2. The expression `r` may mention any arguments to `l` —if `l` is a variational— and may mention the constant `$name` which is the string representation of the name `l` —if `l` is a context declaration.
   - ◇ This is necessary to produce term types, section **??**.

4.4.2 Composition Rule

Variationals v_i may be sequentially applied to a context `p` by writing `p` $\oplus\!\!\rightarrow\ v_1\ \oplus\!\!\rightarrow\ v_2\ \oplus\!\!\rightarrow \cdots \oplus\!\!\rightarrow\ v_n$, which 'threads' the context `p` through each of the variationals —that is, we have forward function application v_n `(`$\cdots$ `(`v_1 `p))`.

$$\frac{p \oplus\!\!\rightarrow\ v \rightsquigarrow q \qquad q \oplus\!\!\rightarrow\ w \rightsquigarrow q}{(p \oplus\!\!\rightarrow\ v) \oplus\!\!\rightarrow\ w \rightsquigarrow q}$$

◇ In the concrete syntax, parenthesis `(,)` are not allowed: $\oplus\!\!\rightarrow$ is left-associative.

$\boxed{\textbf{[MA:}}$ Do we *need* congruence rules for '$\oplus\!\!\rightarrow$'? $\boxed{]}$

4.4.3 Empty Variational Rule

A nullary composition of variationals v_i applied to a context `p` does not alter `p`; i.e., when `n = 0` in `p` $\oplus\!\!\rightarrow$ v_1 $\oplus\!\!\rightarrow$ $\cdots$ $\oplus\!\!\rightarrow$ v_n we have `p` $\oplus\!\!\rightarrow$ which is the same as `p`. Using `Id` from section 4.4.1, we may characterise the identity variational as follows.

$$\frac{}{p \oplus\!\!\rightarrow \mathsf{Id} \rightsquigarrow p}$$

In the concrete syntax, Id is simply whitespace; whence we have the following optimisation laws.

```
p ⊕→ ≈ p
p ⊕→ v ≈ p v ⊕→ ≈ p v
```

In particular, *single variational application* may be written with or without the use of $\oplus\!\!\rightarrow$. Moreover, any variational v that takes an argument of type τ can be thought of as a **binary context-value operator**,

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

4.4.4 :kind , :waist , and :level Rules

The meta-primitive `:kind` declares the tag of a context. If the tag is `PackageFormer` then we have an abstract context that will not directly elaborate into Agda code; otherwise if the tag is **`record, data, module`** —constructs supported by Agda— then we have the following elaboration, where $\mathtt{q}_j$ is the first[5] non-`parameter` qualifier.

[5]The current implementation uses a single 'waist' *number* `j` to identify the first `j`-many parameters.

Kind-Waist Elaboration

```
    Name = ⟨k; ℓ; q_i η_i : τ_i ≔ δ_i⟩_i
⇝
    k Name (η_0 : τ_0 ≔ δ_0) ⋯ (η_{j−1} : τ_{j−1} ≔ δ_{j−1}) : Set ℓ where
      q_j η_j : τ_j
          η_j = δ_j

      q_{j+1} η_{j+1} : τ_{j+1}
              η_{j+1} = δ_{j+1}

      ⋮

      q_k η_k : τ_k
          η_k = δ_k
```

Notice that unless the first `j`-many elements have **no definitions**, the resulting elaboration will result in invalid Agda. Rather than impose a particular way to handle definitional extensions, it is left to the variational designer to handle this —e.g., by performing *'definitional erasure'* or dropping those particular elements.

$$\frac{}{\langle k; \ell; q_i\, n_i : \tau_i \coloneqq d_i\rangle_i \oplus\!\!\longrightarrow :\mathsf{kind}\ k' \rightsquigarrow \langle k'; \ell; q_i\, n_i : \tau_i \coloneqq d_i\rangle_i}$$

We then quickly have *kind-fusion*: `p ⊕⟶ :kind k`$_1$ `⊕⟶ :kind k`$_2$ $\approx$ `p ⊕⟶ :kind k`$_2$.

For instance, `Empty` below is an abstract context and so has no form using existing Agda syntax, whereas `Empty`^{r} elaborates to a valid Agda phrase.

Example :kind Application

```
PackageFormer Empty : Set where

Empty^r = Empty ⊕⟶ :kind record
{-
record Empty : Set where     -- Equivalently
-}
```

If a `PackageFormer` has some elements, like `Type` below, then this approach crashes.

:kind application is not enough

```
PackageFormer Type : Set₁ where
  Carrier : Set

-- Typeʳ = Type :kind record
{-
record Typeʳ : Set₁ where     -- Equivalently
   Carrier : Set              -- Invalid Agda phrase
-}
```

We thus need a way to alter all elements —e.g., by changing their qualifiers to be `field` or `parameter`. Enter the :waist rule:

$$\frac{q_i' = \text{if } i \leq w \text{ then parameter else } q_i}{\langle k; \ell; q_i\, n_i : \tau_i \coloneqq d_i \rangle_i \;\; \text{:waist}\, w \rightsquigarrow \langle k; \ell; q_i'\, n_i : \tau_i \coloneqq d_i \rangle_i}$$

Example :waist Application

```
Typeʳ = Typeʳ :kind record :waist 1
{-
record Type (Carrier : Set) : Set₁ where    -- Equivalently
-}
```

However, the level of `Typeʳ` is unnecessarily large: `Set` suffices in-place of `Set₁`. The level could have been inferred by inspecting the elements of `Typeʳ`, however, we took the conservative option of leaving it to the reader to alter a level by providing either `inc` or `dec` to increment it or decrement it —our abstract interpreter will be more generic: Any function `f` on levels is acceptable.

$$\frac{f : \text{Level} \to \text{Level}}{\langle k; \ell; q_i\, n_i : \tau_i \coloneqq d_i \rangle_i \;\; \text{:level}\, f \rightsquigarrow \langle k;\; f\,\ell; q_i\, n_i : \tau_i \coloneqq d_i \rangle_i}$$

Example :level Application

```
Typeʳ' = Typeʳ :kind record :waist 1 :level dec
{-
record Type (Carrier : Set) : Set where    -- Equivalently
-}
```

4.4.5 Altering Elements —Map Rule

The final meta-primitive is `:alter-elements`; it is the 'hammer' that accomplishes most of the work, it takes an arbitrary function `List Element → List Element` which it then applies to the context to obtain a new, possibly ill-formed, context. As such, the rule for it is rather unhelpful.

$$\frac{f \,:\, \mathsf{List\ Element} \to \mathsf{List\ Element}}{\langle k; \ell; es\rangle \;\; \mathsf{:alter{-}elements}\, f \;\rightsquigarrow\; \langle k; \ell; f\ es\rangle}$$

Instead, using `:alter-elements`, we can define a 'safe' traversal variational, **map**, and provide a rule for it.

$$\frac{e'_i = f(e_i)[\mathsf{name}\, e_j \coloneqq \mathsf{name}\,(f e_j)]_j}{\langle k; \ell; e_i\rangle_i \;\mathsf{map}\; f \;\rightsquigarrow\; \langle k; \ell; e'_i\rangle_i}$$

That is, the function `f` is applied to all elements of a context, while propagating all new name changes to subsequent elements.

For practicality, `map` actually takes some optional arguments; such as `:adjoin-retract` and `:adjoin-coretract` to mechanically produce views —record translations— `record {old-name`$_i$ `= new-name`$_i$`}` and `record {new-name`$_i$ `= old-name`$_i$`}` respectively. For example, `q = p map f :adjoin-retract "go"` produces a new context with a new element `go : q` $\to$ `p` which implements the 'old names' of `p` using the symbols of `q`. Whether such translations are meaningful depends on `f`.

Corollaries of Map

```
𝒱-rename f = map (λ e → (map-name (λ n → (funcall f n)) e))

𝒱-decorated by = rename (λ name → (concat name by))
𝒱-co-decorated by = rename (λ name → (concat by name))
𝒱-primed = decorated "’"
𝒱-subscripted₀ = decorated "₀"
-- ⋮
𝒱-subscripted₉ = decorated "₉"
```

Since decoration is invertible, we could have adjoined both a retract and 'co-retract', as follows.

Decoration is invertible

```
𝒱-decorated’ by = map (λ e → (map-name (λ n → (concat n by)) e))
 ↪  :adjoin-coretract "decorate"
```

4.4.6 Summary of Sample Variationals Provided With The System

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 below. The implementations, in the user manual, are discussed along with the associated Lisp code and use cases.

Name	Description
`record`	Reify a PackageFormer as a valid Agda record
`extended-by`	Extend a PackageFormer by a string-“;”-list of declaration
`keeping`	Largest well-formed PackageFormer consisting of a given list of elements
`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
`data`	Reify a PackageFormer as a valid Agda algebraic data type
`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
`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

Below are the **five meta-primitives** from which all variationals are borne, followed by a few 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.

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.
$\oplus\!\!\rightarrow$	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

4.5 Contributions

1. Expressive & extendable specification language for the library developer.
   - ◇ We demonstrate that our meta-primitives permit this below by demonstrating that ubiquitous module combinators can be easily formalised *and* easily used.
   - ◇ E.g., from a theory we can derive its homomorphism type, signature, its termtype, etc; we generate useful constructions inspired from universal algebra.
   - ◇ 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.
2. Unobtrusive and a tremendously simple interface to the end user.
   - ◇ Once a library is developed using (the current implementation of) PackageFormers, the end user only needs to reference the resulting generated Agda, without any knowledge of the existence of PackageFormers.
     - ○ Generated modules are necessarily 'flattened' for typechecking with Agda.
   - ◇ We demonstrate below how end-users can build upon a library by using *one line* specifications, by showing over over 200 specifications of mathematical structures. $\boxed{\textbf{[MA:}}$??? $\boxed{\textbf{]}}$
3. Efficient: Our current implementation processes over 200 specifications in ~3 seconds; yielding typechecked Agda code.
   - ◇ It is the typechecking that takes time.
4. Pragmatic: We demonstrate how common combinators can be defined for library developers, but also how they can be furnished with concrete syntax —inspired by Agda's— 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.
   - ◇ 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 second part of the user manual, we show how to formulate module combinators using a simple and straightforward subset of Emacs Lisp —a terse introduction is provided.

8. We have a categorical structure consisting of PackageFormers as objects and those variationals that are signature morphisms.

Chapter 5

The `Context` Library

The `PackageFormer` framework is a useful tool to experiment with uncommon ways to package things together, but it contradicts our initial philosophy of having a singular lingua franca for a language and its tongues. With the lessons learned from developing `PackageFormer`, we go on in this section to produce `Context`, an *extensible do-it-yourself module system for Agda **within** Agda.*

We will show an automatic technique for unbundling data at will; thereby resulting in *bundling-independent representations* and in *delayed unbundling.* Our contributions are to show:

1. Languages with sufficiently powerful type systems and meta-programming can conflate record and term datatype declarations into one practical interface. In addition, the contents of these grouping mechanisms may be function symbols as well as propositional invariants —an example is shown at the end of Section 5.2. We identify the problem and the subtleties in shifting between representations in Section 5.1.

2. Parameterised records can be obtained on-demand from non-parameterised records (Section 5.2) .

   - ◇ As with $\texttt{Magma}_0$, the traditional approach Gross, Chlipala, and Spivak [GCS14] to unbundling a record requires the use of transport along propositional equalities, with trivial `refl`-exivity proofs. In Section 5.2, we develop a combinator, `_:waist_`, which removes the boilerplate necessary at the type specialisation location as well as at the instance declaration location.

3. Programming with fixed-points of unary type constructors can be made as simple as programming with term datatypes (Section 5.3).

4. Astonishingly, we mechanically regain ubiquitous data structures such as ℕ, `Maybe`, `List` as the term datatypes of simple pointed and monoidal theories (Section 5.4).

As an application, in Section 5.5 we show that the resulting setup applies as a semantics for a declarative pre-processing tool that accomplishes the above tasks, namely `PackageFormer`.

For brevity, and accessibility, a number of definitions are elided and only dashed pseudo-code is presented in this section, with the understanding that such functions need to be extended homomorphically over all possible term constructors of the host language. Enough is shown to communicate the techniques and ideas, as well as to make the resulting library usable. The details, which users do not need to bother with, can be found in the appendices.

5.1 The Problems

Let us begin anew by briefly reviewing the main problems, but this time directly using Agda as the language of discourse.

There are a number of problems, with the number of parameters being exposed being the pivotal concern. To exemplify the distinctions at the type level as more parameters are exposed, consider the following approaches to formalising a dynamical system —a collection of states, a designated start state, and a transition function.

Dynamical Systems

```
record DynamicSystem₀ : Set₁ where
  field
    State : Set
    start  : State
    next   : State → State

record DynamicSystem₁ (State : Set) : Set where
  field
    start : State
    next  : State → State

record DynamicSystem₂ (State : Set) (start : State) : Set where
  field
    next : State → State
```

Each `DynamicSystem`$_i$ is a type constructor of `i`-many arguments; but it is the types of these constructors that provide insight into the sort of data they contain:

Type	Kind
$\texttt{DynamicSystem}_0$	$\texttt{Set}_1$
$\texttt{DynamicSystem}_1$	Π X : Set • Set
$\texttt{DynamicSystem}_2$	Π X : Set • Π x : X • Set

We shall refer to the concern of moving from a record to a parameterised record as **the**

unbundling problem Garillot et al. [Gar+09]. For example, moving from the *type* `Set`$_1$ to the *function type* `Π X : Set • Set` gets us from `DynamicSystem`$_0$ to something resembling `DynamicSystem`$_1$, which we arrive at if we can obtain a *type constructor* `λ X : Set • ⋯`. We shall refer to the latter change as *reïfication* since the result is more concrete: It can be applied. This transformation will be denoted by $\Pi{\rightarrow}\lambda$. To clarify this subtlety, consider the following forms of the polymorphic identity function. Notice that id_i *exposes* `i`-many details at the type level to indicate the sort of data it consists of. However, notice that `id`$_0$ is a type of functions whereas id_1 is a function on types. Indeed, the latter two are derived from the first one: $\mathsf{id}_{i+1} = \Pi{\rightarrow}\lambda\,\mathsf{id}_i$ These identities are true by `refl`-exivity —see Appendix A.8.

Polymorphic Identity Functions

```
id₀ : Set₁
id₀ = Π X : Set • Π e : X • X

id₁ : Π X : Set • Set
id₁ = λ (X : Set) → Π e : X • X

id₂ : Π X : Set • Π e : X • Set
id₂ = λ (X : Set) (e : X) → X
```

Of course, there is also the need for descriptions of values, which leads to term datatypes. We shall refer to the shift from record types to algebraic data types as **the termtype problem**. Our aim is to obtain all of these notions —of ways to group data together— from a single user-friendly context declaration, using monadic notation.

5.2 Monadic Notation

There is little use in an idea that is difficult to use in practice. As such, we conflate records and termtypes by starting with an ideal syntax they would share, then derive the necessary artefacts that permit it. Our choice of syntax is monadic do-notation [Mog91b; Mar+16]:

?

```
DynamicSystem : Context ℓ₁
DynamicSystem = do State ← Set
                   start ← State
                   next  ← (State → State)
                   End
```

Here `Context, End`, and the underlying monadic bind operator are unknown. Since we want to be able to *expose* a number of fields at will, we may take `Context` to be types indexed by a number denoting exposure. Moreover, since records are product types, we expect there to be a recursive definition whose base case will be the identity of products, the unit type $\mathbb{1}$

—which corresponds to ⊤ in the Agda standard library and to `()` in Haskell.

Exposure	Elaboration
0	`Σ State : Set • Σ start : X • Σ next : State → State • 𝟙`
1	`Π State : Set • Σ start : X • Σ next : State → State • 𝟙`
2	`Π State : Set • Π start : X • Σ next : State → State • 𝟙`
3	`Π State : Set • Π start : X • Π next : State → State • 𝟙`

Table 5.1: Elaborations of DynamicSystem at various exposure levels

With these elaborations of `DynamicSystem` to guide the way, we resolve two of our unknowns.

Context and End

```
{- “Contexts” are exposure-indexed types -}
Context = λ ℓ → ℕ → Set ℓ

{- Every type can be used as a context -}
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S

{- The “empty context” is the unit type -}
End : ∀ {ℓ} → Context ℓ
End = ‘ 𝟙
```

It remains to identify the definition of the underlying bind operation `>>=`. Usually, for a type constructor `m`, bind is typed `∀ {X Y : Set} → m X → (X → m Y) → m Y`. It allows one to “extract an `X`-value for later use” in the `m Y` context. Since our `m = Context` is from levels to types, we need to slightly alter bind’s typing.

Defining Bind —First Attempt

```
_>>=_ : ∀ {a b}
      → (Γ : Context a)
      → (∀ {n} → Γ n → Context b)
      → Context (a ⊎ b)
(Γ >>= f) zero    = Σ γ : Γ 0 • f γ 0
(Γ >>= f) (suc n) = Π γ : Γ n • f γ n
```

The definition here accounts for the current exposure index: If zero, we have *record types*, otherwise *function types*. Using this definition, the above dynamical system context would need to be expressed using the lifting quote operation. **The extensibility is provided by the definition of bind: Rather than Σ and Π, users may use or augment the framework in other forms.**

Example Use

```
‘ Set >>= λ State
       → ‘ State >>= λ start
                 → ‘ (State → State) >>= λ next
                                     → End

{- or -}

do State ← ‘ Set
   start ← ‘ State
   next  ← ‘ (State → State)
   End
```

Interestingly Bird [Bir09] and Hudak et al. [Hud+07], use of `do`-notation in preference to bind, ≫=, was suggested by John Launchbury in 1993 and was first implemented by Mark Jones in Gofer. Anyhow, with our goal of practicality in mind, we shall "build the lifting quote into the definition" of bind:

The Definition of Bind

```
_>>=_ : ∀ {a b}
      → (Γ : Set a)  -- Main difference
      → (Γ → Context b)
      → Context (a ⊎ b)
(Γ >>= f) zero    = Σ γ : Γ • f γ 0
(Γ >>= f) (suc n) = Π γ : Γ • f γ n
```

With this definition, the above declaration `DynamicSystem` typechecks. However, `DynamicSystem` $i \not\cong$ `DynamicSystem`$_i$, instead `DynamicSystem` i are "factories": Given `i`-many arguments, a product value is formed. What if we want to *instantiate* some of the factory arguments ahead of time?

Factories and Instantiation

```
𝒩₀ : DynamicSystem 0   {- See the elaborations in Table 1 -}
𝒩₀ = ℕ , 0 , suc , tt

𝒩₁ : DynamicSystem 1
𝒩₁ = λ State → ??? {- Impossible to complete if "State" is empty! -}

{- "Instantiaing" X to be ℕ in "DynamicSystem 1" -}
𝒩₁' : let State = ℕ in Σ start : State  • Σ s : (State → State)  • 𝟙
𝒩₁' = 0 , suc , tt
```

It seems what we need is a method, say Π→λ, that takes a Π-type and transforms it into a λ-expression. One could use a universe, an algebraic type of codes denoting types, to define Π→λ. However, one can no longer then easily use existing types since they are not formed from the universe's constructors, thereby resulting in duplication of existing types via the universe encoding. This is neither practical nor pragmatic.

As such, we are left with pattern matching on the language's type formation primitives as the only reasonable approach. The method Π→λ is thus a macro[1] that acts on the syntactic term representations of types. Below is the main transformation —the details can be found in Appendix A.7.

```
Π→λ (Π a : A • τ) = (λ a : A • τ)
```

That is, we walk along the term tree replacing occurrences of Π with λ. For example,

Example use of Π→λ

```
  Π→λ (Π→λ (DynamicSystem 2))
≡{- Definition of DynamicSystem at exposure level 2 -}
  Π→λ (Π→λ (Π X : Set • Π s : X  • Σ n : X → X  • 𝟙))
≡{- Definition of Π→λ -}
  Π→λ (λ X : Set • Π s : X  • Σ n : X → X  • 𝟙)
≡{- Homomorphy of Π→λ -}
  λ X : Set • Π→λ (Π s : X  • Σ n : X → X  • 𝟙)
≡{- Definition of Π→λ -}
  λ X : Set • λ s : X  • Σ n : X → X  • 𝟙
```

For practicality, _:waist_ is a macro (defined in Appendix A.9) acting on contexts that repeats Π→λ a number of times in order to lift a number of field components to the parameter level.

```
τ :waist n = Π→λ^n (τ n)
f^0 x      = x
f^{n+1} x  = f^n (f x)
```

We can now "fix arguments ahead of time". Before such demonstration, we need to be mindful of our practicality goals: One declares a grouping mechanism with `do ... End`, which in turn has its instance values constructed with ⟨ ... ⟩.

[1]A *macro* is a function that manipulates the abstract syntax trees of the host language. In particular, it may take an arbitrary term, shuffle its syntax to provide possibly meaningless terms or terms that could not be formed without pattern matching on the possible syntactic constructions. An up to date and gentle introduction to reflection in Agda can be found at [Alh19a]

Syntactic Sugar for Context Values

```
-- Expressions of the form “⋯ , tt” may now be written “⟨ ⋯ ⟩”
infixr 5 ⟨ _⟩
⟨⟩ : ∀ {ℓ} → 𝟙 {ℓ}
⟨⟩ = tt

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

_⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × (𝟙 {ℓ})
s ⟩ = s , tt
```

The following instances of grouping types demonstrate how information moves from the body level to the parameter level.

Unbundling: Lifting Fields into Parameters

```
𝒩⁰ : DynamicSystem :waist 0
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : (DynamicSystem :waist 1) ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : (DynamicSystem :waist 2) ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : (DynamicSystem :waist 3) ℕ 0 suc
𝒩³ = ⟨⟩
```

Using `:waist` i we may fix the first i-parameters ahead of time. Indeed, the type `(DynamicSystem :waist 1) ℕ` is *the type of dynamic systems over carrier* ℕ, whereas `(DynamicSystem :waist 2) ℕ 0` is *the type of dynamic systems over carrier* ℕ *and start state 0.*

Examples of the need for such on-the-fly unbundling can be found in numerous places in the Haskell standard library. For instance, the standard libraries *Haskell Basic Libraries — Data.Monoid* [20] have two isomorphic copies of the integers, called `Sum` and `Product`, whose reason for being is to distinguish two common monoids: The former is for *integers with addition* whereas the latter is for *integers with multiplication*. An orthogonal solution would be to use contexts:

?

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              _⊕_     ← (Carrier → Carrier → Carrier)
              Id      ← Carrier
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

With this context, (`Monoid` ℓ_0 `:waist 2) M ⊕` is the type of monoids over *particular* types `M` and *particular* operations ⊕. Of-course, this is orthogonal, since traditionally unification on the carrier type `M` is what makes typeclasses and canonical structures Mahboubi and Tassi [MT13] useful for ad-hoc polymorphism.

5.3 Termtypes as Fixed-points

We have a practical monadic syntax for possibly parameterised record types that we would like to extend to termtypes. Algebraic data types are a means to declare concrete representations of the least fixed-point of a functor; see Swierstra [Swi08] for more on this idea. In particular, the description language 𝔻 for dynamical systems, below, declares concrete constructors for a fixpoint of a certain functor `F`; i.e., 𝔻 ≅ `Fix F` where:

ADTs and Functors

```
data 𝔻 : Set where
    startD : 𝔻
    nextD  : 𝔻 → 𝔻

F : Set → Set
F = λ (D : Set) → 𝟙 ⊎ D

data Fix (F : Set → Set) : Set where
  μ : F (Fix F) → Fix F
```

The problem is whether we can derive `F` from `DynamicSystem`. Let us attempt a quick calculation sketching the necessary transformation steps (informally expressed via “⇒”):

From Contexts to Fixed-points

```
  do S ← Set; s ← S; n ← (S → S); End
⇒ {- Use existing interpretation to obtain a record. -}
  Σ S : Set • Σ s : S • Σ n : (S → S) • 𝟙
⇒ {- Pull out the carrier, “:waist 1”,
   to obtain a type constructor using “Π→λ”. -}
  λ S : Set • Σ s : S • Σ n : (S → S) • 𝟙
⇒ {- Termtype constructors target the declared type,
   so only their sources matter. E.g., ‘s : S’ is a
   nullary constructor targeting the carrier ‘S’.
   This introduces 𝟙 types, so any existing
   occurances are dropped via 𝟘. -}
  λ S : Set • Σ s : 𝟙 • Σ n : S • 𝟘
⇒ {- Termtypes are sums of products. -}
  λ S : Set •       𝟙    ⊎      S    ⊎ 𝟘
⇒ {- Termtypes are fixpoints of type constructors. -}
  Fix (λ X • 𝟙 ⊎ S)  -- i.e., 𝔻
```

Since we may view an algebraic data-type as a fixed-point of the functor obtained from the union of the sources of its constructors, it suffices to treat the fields of a record as constructors, then obtain their sources, then union them. That is, since algebraic-datatype constructors necessarily target the declared type, they are determined by their sources. For example, considered as a unary constructor `op : A → B` targets the termtype `B` and so its source is `A`. The details on the operations ↓↓, Σ→⊎, and `sources` characterised by the pseudocode below can be found in appendices A.3.4, A.12.4, and A.12.3, respectively. It suffices to know that Σ→⊎ rewrites dependent-sums into disjoint sums, which requires the second argument to lose its reference to the first argument which is accomplished by ↓↓; further details can be found in the appendices.

```
↓↓ τ = “reduce all de Bruijn indices within τ by 1”

Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (↓↓ Ba)

sources (λ x : (Π a : A • Ba) • τ) = (λ x : A • sources τ)
sources (λ x : A              • τ) = (λ x : 𝟙 • sources τ)

termtype τ = Fix (Σ→⊎ (sources τ))
```

It is instructive to work through the process of how 𝔻 is obtained from `termtype` in order to demonstrate that this approach to algebraic data types is practical **within Agda**.

Declaring a Derived Termtype

```
𝔻 = termtype (DynamicSystem :waist 1)

-- Pattern synonyms for more compact presentation
pattern startD  = μ (inj₁ tt)       -- : 𝔻
pattern nextD e = μ (inj₂ (inj₁ e)) -- : 𝔻 → 𝔻
```

With these `pattern` declarations, we can actually use the more meaningful names **`startD`** and **`nextD`** when pattern matching, instead of the seemingly daunting μ-**`inj`**-ections. For instance, we can immediately see that the natural numbers act as the description language for dynamical systems:

Seemingly Trivial Remappings

```
to : 𝔻 → ℕ
to startD    = 0
to (nextD x) = suc (to x)

from : ℕ → 𝔻
from zero    = startD
from (suc n) = nextD (from n)
```

Readers whose language does not have `pattern` clauses need not despair. With the macro

```
Inj n x = μ (inj₂ ⁿ (inj₁ x))
```

we may define `startD = Inj 0 tt` and `nextD e = Inj 1 e` —that is, constructors of termtypes are particular injections into the possible summands that the termtype consists of. Details on this macro may be found in appendix A.12.6.

5.4 Free Datatypes from Theories

Astonishingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if a parameterised context $\mathcal{C}$ `: Set → Context` ℓ_0 is given, then

Abstract Syntax Trees

```
ℂ = λ X → termtype (𝒞 X :waist 1)
```

can be used to form ‘free, lawless, $\mathcal{C}$-instances’. For instance, earlier we witnessed that the termtype of dynamical systems is essentially the natural numbers.

Theory	Termtype
Dynamical Systems	ℕ
Pointed Structures	Maybe
Monoids	Binary Trees

Table 5.2: Data structures as free theories

The final entry in Table 2 is a well known correspondence that we can now not only formally express, but also prove to be true.

Trees from Monoids

```
𝕄 : Set
𝕄 = termtype (Monoid ℓ₀ :waist 1)
{- i.e., Fix (λ X → 𝟙         -- Id, nil leaf
               ⊎ X × X × 𝟙 -- _⊕_, branch
               ⊎ 𝟘          -- invariant leftId
               ⊎ 𝟘          -- invariant rightId
               ⊎ X × X × 𝟘 -- invariant assoc
               ⊎ 𝟘)         -- the "End {ℓ}"
-}

-- Pattern synonyms for more compact presentation
pattern emptyM      = μ (inj₂ (inj₁ tt))               -- : 𝕄
pattern branchM l r = μ (inj₁ (l , r , tt))            -- : 𝕄 → 𝕄 → 𝕄
pattern absurdM a   = μ (inj₂ (inj₂ (inj₂ (inj₂ a)))) -- absurd 𝟘-values

data TreeSkeleton : Set where
  empty  : TreeSkeleton
  branch : TreeSkeleton → TreeSkeleton → TreeSkeleton
```

Using Agda's Emacs interface, we may interactively case-split on values of 𝕄 until the declared patterns appear, then we associate them with the constructors of `TreeSkeleton`.

Seemingly Trivial Remappings

```
to : 𝕄 → TreeSkeleton
to emptyM        = empty
to (branchM l r) = branch (to l) (to r)
to (absurdM (inj₁ ()))
to (absurdM (inj₂ ()))

from : TreeSkeleton → 𝕄
from empty        = emptyM
from (branch l r) = branchM (from l) (from r)
```

That these two operations are inverses is easily demonstrated.

Trees from Monoids

```
fromoto : ∀ m → from (to m) ≡ m
fromoto emptyM        = refl
fromoto (branchM l r) = cong₂ branchM (fromoto l) (fromoto r)
fromoto (absurdM (inj₁ ()))
fromoto (absurdM (inj₂ ()))

tooofrom : ∀ t → to (from t) ≡ t
toofrom empty        = refl
toofrom (branch l r) = cong₂ branch (toofrom l) (toofrom r)
```

Without the `pattern` declarations the result would remain true, but it would be quite difficult to believe in the correspondence without a machine-checked proof.

To obtain a data structure over some 'value type' Ξ, one must start with "theories containing a given set Ξ". For example, we could begin with the theory of abstract collections, then obtain lists as the associated termtype.

Lists from Paramterised Collections

```
Collection : ∀ ℓ → Context (ℓsuc ℓ)
Collection ℓ = do Elem    ← Set ℓ
                  Carrier ← Set ℓ
                  insert  ← (Elem → Carrier → Carrier)
                  ∅       ← Carrier
                  End {ℓ}

ℂ : Set → Set
ℂ Elem = termtype ((Collection ℓ₀ :waist 2) Elem)

pattern _::_ x xs = μ (inj₁ (x , xs , tt))
pattern  ∅        = μ (inj₂ (inj₁ tt))
```

Realising Collection ASTs as Lists

```
to : ∀ {E} → ℂ E → List E
to (e :: es) = e :: to es
to ∅         = []
```

It is then little trouble to show that `to` is invertible. We invite the readers to join in on the fun and try it out themselves!

5.5 Related Works

Surprisingly, conflating parameterised and non-parameterised record types with termtypes *within a language in a practical fashion* has not been done before.

The `PackageFormer` Al-hassy, Carette, and Kahl [ACK19] and Al-hassy [Alh19b] editor extension reads contexts —in nearly the same notation as `Context`— enclosed in dedicated comments, then generates and imports Agda code from them seamlessly in the background whenever typechecking happens. The framework provides a fixed number of meta-primitives for producing arbitrary notions of grouping mechanisms, and allows arbitrary Emacs Lisp Graham [Gra95] to be invoked in the construction of complex grouping mechanisms.

	PackageFormer	Contexts
Type of Entity	Preprocessing Tool	Language Library
Specification Language	Lisp + Agda	Agda
Well-formedness Checking	×	✓
Termination Checking	✓	✓
Elaboration Tooltips	✓	×
Rapid Prototyping	✓	✓ (Slower)
Usability Barrier	None	None
Extensibility Barrier	Lisp	Weak Metaprogramming

Table 5.3: Comparing the in-language `Context` mechanism with the `PackageFormer` editor extension

The `PackageFormer` paper Al-hassy, Carette, and Kahl [ACK19] provided the syntax necessary to form useful grouping mechanisms but was shy on the semantics of such constructs. We have chosen the names of the `Context` combinators to closely match those of `PackageFormer`'s with an aim of furnishing the mechanism with semantics by construing the syntax as semantics-functions; i.e., we have a shallow embedding of `PackageFormer`'s constructs as Agda entities:

Syntax	Semantics
`PackageFormer`	`Context`
`:waist`	`:waist`
⊕→	Forward function application
`:kind`	`:kind`, see below
`:level`	Agda built-in
`:alter-elements`	Agda macros

Table 5.4: `Context` as a semantics for `PackageFormer` constructs

`PackageFormer`'s `_:kind_` meta-primitive dictates how an abstract grouping mechanism should be viewed in terms of existing Agda syntax. However, unlike PackageFormer, all of our syntax consists of legitimate Agda terms. Since language syntax is being manipulated, we are forced to implement the `_:kind_` meta-primitive as a macro —further details can be found in Appendix A.13.

Codes for Agda's First-class Grouping Mechanisms

```
data Kind : Set where
  ‘record    : Kind
  ‘typeclass : Kind
  ‘data      : Kind
```

```
𝒞 :kind ‘record = 𝒞 0
𝒞 :kind ‘typeclass = 𝒞 :waist 1
𝒞 :kind ‘data = termtype (𝒞 :waist 1)
```

We did not expect to be able to define a full Agda implementation of the semantics of `PackageFormer`'s syntactic constructs due to Agda's rather constrained metaprogramming mechanism. However, it is important to note that `PackageFormer`'s Lisp extensibility expedites the process of trying out arbitrary grouping mechanisms —such as partial-choices of pushouts and pullbacks along user-provided assignment functions— since it is all either string or symbolic list manipulation. On the Agda side, using `Context`, it would require substantially more effort due to the limited reflection mechanism and the intrusion of the stringent type system.

5.6 Conclusion

Starting from the insight that related grouping mechanisms could be unified, we showed how related structures can be obtained from a single declaration using a practical interface. The resulting framework, based on contexts, still captures the familiar record declaration syntax as well as the expressivity of usual algebraic datatype declarations —at the minimal cost of using `pattern` declarations to aide as user-chosen constructor names. We believe that our approach to using contexts as general grouping mechanisms *with* a practical interface are interesting contributions.

We used the focus on practicality to guide the design of our context interface, and provided interpretations both for the rather intuitive "contexts are name-type records" view, and for the novel "contexts are fixed-points" view for termtypes. In addition, to obtain parameterised variants, we needed to explicitly form "contexts whose contents are over a given ambient context" —e.g., contexts of vector spaces are usually discussed with the understanding that there is a context of fields that can be referenced— which we did using the name binding machanism of `do`-notation. These relationships are summarised in the following table.

Concept	Concrete Syntax	Description
Context	`do S ← Set; s ← S; n ← (S → S); End`	“name-type pairs”
Record Type	`Σ S : Set • Σ s : S • Σ n : S → S • 𝟙`	“bundled-up data”
Function Type	`Π S • Σ s : S • Σ n : S → S • 𝟙`	“a type of functions”
Type constructor	`λ S • Σ s : S • Σ n : S → S • 𝟙`	“a function on types”
Algebraic datatype	`data 𝔻 : Set where s : 𝔻; n : 𝔻 → 𝔻`	“a descriptive syntax”

Table 5.5: Contexts embody all kinds of grouping mechanisms

To those interested in exotic ways to group data together —such as, mechanically deriving product types and homomorphism types of theories— we offer an interface that is extensible using Agda’s reflection mechanism. In comparison with, for example, special-purpose preprocessing tools, this has obvious advantages in accessibility and semantics.

To Agda programmers, this offers a standard interface for grouping mechanisms that had been sorely missing, with an interface that is so familiar that there would be little barrier to its use. In particular, as we have shown, it acts as an in-language library for exploiting relationships between free theories and data structures. As we have only presented the high-level definitions of the core combinators, leaving the Agda-specific details to the appendices, it is also straightforward to translate the library into other dependently-typed languages.

Chapter 6

Discussion, Conclusion, and Future Work

As discussed in the introduction, Section 1.3, there is a lack of a unified and practical language for dealing with grouping mechanisms as standard and unexceptional *values*. In an effort to address this issue, this thesis has presented a framework for modelling grouping mechanisms as first class values. This framework aids in advancing the current understanding of grouping mechanisms —as summarised in the table below.

Grouping Mechanism	Distinguishing Features
`record`	Opaque *fields*
`data`	Uninterpreted *constructors*
`module`	Namespacing for derived results or definitional extensions
`Context`	Mixture of fields, constructors, and derived results

This chapter summarises and discusses the contributions of this thesis and points to future research directions resulting from this work. Specifically, Section 6.1 highlights, discuss, and assesses the contributions that are made by this thesis. Section 6.2 suggests avenues for future work resulting from the prototypical `PackageFormer` framework, and its applications and tools. Finally, Section 6.3 makes final comments and closing remarks.

6.1 Highlights of the Contributions

The contributions related to the proposed framework for the modelling grouping mechanisms as first class values include:

1. **Necessary conditions for the first-class grouping mechanisms:**

   This thesis proposed a set of necessary conditions for a programming language to treat grouping mechanisms as ordinary values. If the language provides a fixed-point operator and metaprogramming, then algebraic data types can be obtained from contexts. The minimal number of constraints aid in advancing the current understanding of grouping mechanisms by serving as a basis permitting exotic notions of grouping to be experimented with —such as soundly deriving homomorphism types from a given context declaration.

2. **Specification of the relationships between grouping mechanisms:**

   This thesis builds atop well-established foundations but *without dictatorially declaring* what grouping mechanisms are 'useful' and which should be omitted. Instead, we provide primitives from which new grouping notions may be derived. This is akin to a programming language providing a set of combinators from which complex programs may be constructed.

3. **Two tools to experiment with new notions of grouping mechanisms:**

   This thesis provides the dynamic `PackageFormer` Emacs extension for *rapid, albeit practical, prototyping* of new notions of grouping mechanisms. For example, as discussed in Section 4.3.4, partial choices of pushouts can be quickly expressed and usefully so. In contrast, the thesis also provides the `Context` library for *sound, typechecked,* developments of new notions of grouping mechanisms.

4. **Two semantics for a practical and pragmatic tool for developing grouping mechanisms:**

   This thesis provides a rewrite-based semantics for the `PackageFormer` Emacs editor extension, as well as interpreting its syntax as semantically as Agda-functions via the correspondence with `Context` outlined in Section 5.5.

5. **Numerous examples of common grouping mechanisms:**

   This thesis presented a number of way to group data together, in the setting of `PackageFormer` in Section 4.3, with many more formalised in the tool's online repository[1]. The usability of these notions of grouping mechanisms were found by exploring existing libraries on dependently typed languages, in Section 2, which then led us to provide a listing of 'design patterns for dependently typed languages'.

6.2 Future Work

The frameworks presented in this thesis can be extended in a number of different directions. The following subsections describe possible extensions and further work with respect to the proposed framework and applications thereof.

[1] https://alhassy.github.io/next-700-module-systems/prototype/package-former.html

6.2.1 Theory: Models and Techniques

Concerning the proposed framework for capturing the relationships of grouping mechanisms, the following directions can be explored:

1. An investigation into providing more, and potentially better semantical models, of `PackageFormer` ought to be undertaken. For instance, since grouping mechanisms are about organising data and we wish to treat them as first-class citizens, a natural semantics would be using a form of closed categories, institutions, or rooms and corridors [FM93; Dia08; Shu16; Mai05; Cas+15; Mog91a].

2. Further investigation into the interplay between external and internal grouping mechanisms. For instance, in Agda record projections can be costly [Per17] but every record-valued function is essentially a parameterised module, so an automatic shift in tools could result in increased efficiency.

3. A further study into the relationships between the module primitives and the ambient type theories is needed. For example, one can study how different λ-calculi can be used in place of the MLTT-fragment in our initial semantics for `PackageFormer`. This can allow for flexible representations and varying degrees of expressivity. Moreover, such an investigation can allow for different way to reason about modules, such as using a little theories approach [FGJ92].

4. There are a number of directions that can be explored with regard to a re-implementation of the `Context` library. For example, sufficiently careful definitions could be used to mitigate the need to side-step Agda's termination checker. Moreover, rather than leap to `do`-notation there is an interesting class of modules that could be defined using an applicative structure —a sort-of Cartesian functor. Such modules would consist of entities that are defined independently of each other and thus the order of their declaration is essentially irrelevant, and as such could provide an opportunity for optimisation.

5. With regard to the existing representation of `Context`, there are a number of possible extensions. For example, rather than simply using Σ and Π based on whether the exposure is zero or not, what if the library were parameterised by a *family* of such 'binding quantifiers', say $\bigoplus$, and for each level of exposure n we used the quantifier $\bigoplus_n$. For instance, we may be interested in interleaving arbitrary meets and joins of a complete lattice rather than just consider types.

6. The `Context` semantics to `PackageFormer`'s `:alter-elements` sledgehammer is the notion of Agda macros. It would be ideal to consider well-behaved approximations to this primitive that also have well-behaved semantical counterparts. This would necessitate an investigation into what people actually want to do to the components of a module and how to do so coherently. From our experience of implementing multiple notions of modules, we feel that `p :alter-elements f` will likely be a colimit construction so as to account for possibly 'dangling edges' when deletion happens and to also account for 'gluing components' when items are identified.

7. The material presented in Chapter 5 4.3 can serve as the basis for many future research directions into *mechanising* notions of grouping mechanisms. In particular, work can be done in order to further articulate how constructions from universal algebra can be used to produce useful notions of grouping mechanisms.

6.2.2 Applications

With respect to the possible applications of the proposed framework, the following directions can be investigated further.

1. The range of the application domain of `Context` can be explored further. Such domains include the λ-calculi of programming languages with varying degrees of type expressivity —-from unityped to dependently-typed languages. This can lead to new and innovative ways to think and reason about modular programming in such systems, as well as to provide insight into the interplay between seemingly different notions of modularisation. For example, by modelling the C language's `#include` approach to modules using `Context`, it may be possible to mechanically derive algebraic data types which may act as a 'description language' for the purposes of serialisation.

2. A study of how the proposed framework can be used to support the treatment of modules as standard values *derivable* from other values is another fruitful future research opportunity. For example, a constant is essentially a module consisting of one entity, whereas a function is essentially a parameterised module. It would be interesting to see which properties of functions could be transformed into ideas about modules; here is a preliminary sketch.

Functional concept	Modular counterpart
Application	Parameter instantiation
λ-abstraction	Module formation
Currying	Module nesting
Continuations	???

   Continuations, also known as 'generalised double negation', can be used to provide structural control-flow and, specialised as difference lists, provide efficiency gains. Perhaps similar benefits lie on the side of modules.

3. An exploration into how the proposed framework could be used to 'discover' modular patterns in existing systems for the purposes of releasing modular libraries *after* an initial library has been written. This could allow library developers to rapidly produce systems with, say, 'specification hints' to guide the automation of discovering certain kinds of packaging structures and hierarchical organisations.

4. Further investigations into how contexts can be developed such that the resulting interplay of contexts, within a system, exhibit a set of desirable properties, such as well-behaved cyclic dependencies. This can lead to new and innovative ways to construct modules. In conjunction with the previous item, this can be used in new applications of modules such as large-scale refactoring of libraries, say, by requesting the pullback, 'intersection', of two libraries to find their well-behaved, implicitly shared, common parent module. More concretely, in object-oriented programming, this is tantamount to taking two classes `Q, R` and requesting they be re-factored by introducing a new parent `P` class with children classes `Q', R'` which are *observationally indistinguishable* from `Q, R`.

6.2.3 Tools and Automation

This thesis developed a prototype tool, `PackageFormer`, to mitigate the amount of duplication present in designing a library by declaratively specifying how new packages are to be formed from existing ones. Moreover, the `Context` library has been developed in order to provide a type-checked analogue; thereby, showing that the treatment of modular values is promising in languages with sufficient power. There are a number of ways in which these tools can be enhanced and extended to provide a more comprehensive tool for specifying packages.

1. The prototype tool can be enhanced by further testing. In particular, further work into investigating ways in which the tool can be used to model commonly occurring notions of packaging as inspired by universal algebra and category theory, such as forming products and limits, then proving coherence results.

2. The prototype tool can be extended by incorporating the functionality to support 'discovarability'. Currently, the prototype can be programmed by the user to form intersections of packages, but this only results in a new package rather than in a new *library*. Given one library, we would like to produce a new library where discovered shared packages are factored-out.

3. The prototype provides rapid development whereas the `Contenxt` library gives rise to analysis and verification, as such it would be beneficial for the prototype tool to interface directly with `Context`.

6.3 Closing Remarks

The treatment of packages as unexceptional values is an ongoing and ambitious endeavour, particular with the respect to the increasing connectedness and complexity of large software developments. A reduction in the number of tongues that a user needs to be familiar with in a programming language reduces accessibility barriers, thereby reducing the amount of repetitive code.

Using the little theories approach [FGJ92] to software development can mitigate duplication of code. However, it is unreasonable and impractical to always enforce a disciplined and fine-grained approach to developing software. Instead, the prototype allows *after the fact* library refactoring by specifying relationships between grouping mechanisms —as demonstrated in Section 4.3.6.

```
#+latex_header_extra: \newglossaryentry{module-systems}{name={Module Systems}, descripti
= t$ where $t$ may refer to all names declared in $Q$. That is, a substition is a map of
```

Glossary

Context A sequence of "variable : type [:= definition]" declarations; a dictionarry associating variables to types and, optionally, a definition; c.f., record-type and object-oriented class; see 'JSON Object'. 33, 74

Curry-Howard Correspondence Programming and proving are essentially the same idea. 14

Dependent Function A function whose result type depends on the value of the argument. 21

Do-Notation Syntactic abbrevation that renders purely functional code as if it were sequential and imperative.. 27

Homoiconic The lack of distinction between 'data' and 'method'. E.g., `'(+ 1 2)` is considered a list of symbols, whereas the *unquoted* term `(+ 1 2)` is considered a function call that reduces to 3. 2

Interpretation See 'Substitution'. 76

JSON object A comma-separted list of key-value pairs; an alias for 'dictionary', 'hashmap', and 'object'. 70

Little Theories The dicipline of building a library by adding one new orthogonal feature at each stage of the hierarchy; c.f., the Interface Segregation Principle. 72

Mixin The ability to extend a datatype with additional functionality long after, and far from, its definition. See also typeclass. Mixins could be simulated as module-to-module functions, which give rise to 'a module of type M' as an instance of the mixin M; e.g., a type of type Show is an instance of the typeclass Show. 71

Module Systems Module systems parameterise programs, proofs, and tactics over structures. They come in many flavours that each communicate a utility difference; e.g., tuples for quickly returning multiple values from a function, a record to treate pieces as a coherent whole, a function as an indexed value, and paramterised modules which 'build upon' other coherent units. 36, 73

Record Rather than holding a bunch of items in our hands and running around with them, we can put them in a bag and run around with it. That is, a record type bundles up related concepts so that may be treated as one coherent entity. If record types can 'inherit' from one another, then we have the notion of an 'object'. 2

Signature A sequence of pairs of name-type declarations; an alias for 'context' and 'telescope'; see also JSON Object and Theory Presentation. 43

Substitution A typed-substition of kind $P \to Q$, also known as a 'view' or 'theory morphism', is a context δ such that every P-delcaration $x : \tau$ has an associated δ-declaration $x \coloneqq t$ where t may refer to all names declared in Q. That is, a substition is a map of contexts that implements the interface of the source using utilities of the target; whence it gives rise to a type-preseving homomorphism on terms which —using the propositions-as-types correspondecnece— preserves truthhood of results. For instance if I implements 'interface' (context) P which can be viewed as Q, then I can be viewed as an implementation of Q. 76

Theory Morphism See 'Substitution'. 74

Theory Presentation A (named) list of name-type declarations, where the type may be a formulae that governs how earlier declared names are inteded to interact. Essentially, it is a signature in a DTL. 73

Typeclass Essentially a dictionary that associates types with a particular list of methods which define the typeclass. Whenever such a method is invoked, the dictionary is accessed for the inferred type and the appropriate definition is used, if possible. This provides a form of ad-hoc polymorphism: We have a list of methods that appear polymorphic, but in-fact their definitions depend on a particular parent type. 2

View See 'Substitution'. 75

[WK: Don't cite Wikipedia in your PhD thesis, and don't copy anything from Wikipedia. Just don't. **]**

Bibliography

[18a] *Curry–Howard correspondence — Wikipedia, The Free Encyclopedia*. 2018. URL: `https://en.wikipedia.org/wiki/Curry-Howard_correspondence` (visited on 10/16/2018) (cit. on p. 62).

[18b] *Hungarian notation — Wikipedia, The Free Encyclopedia*. 2018. URL: `https://en.wikipedia.org/wiki/Hungarian_notation` (visited on 10/16/2018) (cit. on p. 55).

[20] *Haskell Basic Libraries — Data.Monoid*. 2020. URL: `http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Monoid.html` (visited on 03/03/2020) (cit. on p. 111).

[ACK19] 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` (cit. on p. 117).

[Alh19a] Musa Al-hassy. *A slow-paced introduction to reflection in Agda —Tactics!* 2019. URL: `https://github.com/alhassy/gentle-intro-to-reflection` (cit. on p. 110).

[Alh19b] Musa Al-hassy. *The Next 700 Module Systems: Extending Dependently-Typed Languages to Implement Module System Features In The Core Language*. 2019. URL: `https://alhassy.github.io/next-700-module-systems-proposal/thesis-proposal.pdf` (cit. on p. 117).

[Alt] Thorsten Altenkirch. *Inconsistency of Set:Set*. URL: `http://www.cs.nott.ac.uk/~psztxa/g53cfr/l20.html/l20.html` (visited on 10/19/2018) (cit. on p. 16).

[Ast+02] Egidio Astesiano et al. "CASL: the Common Algebraic Specification Language". In: *Theor. Comput. Sci.* 286.2 (2002), pp. 153–196. DOI: `10.1016/S0304-3975(01)00368-1`. URL: `https://doi.org/10.1016/S0304-3975(01)00368-1` (cit. on p. 22).

[BAT14] Gavin M. Bierman, Martı′n Abadi, and Mads Torgersen. “Understanding TypeScript”. In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. 2014, pp. 257–281. DOI: 10.1007/978-3-662-44202-9_11. URL: https://doi.org/10.1007/978-3-662-44202-9%5C_11 (cit. on p. 57).

[BD08] Ana Bove and Peter Dybjer. “Dependent Types at Work”. In: *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*. 2008, pp. 57–99. DOI: 10.1007/978-3-642-03153-3_2. URL: https://doi.org/10.1007/978-3-642-03153-3%5C_2 (cit. on p. 155).

[BG13] Jean-Philippe Bernardy and Moulin Guilhem. “Type-theory in Color”. In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 61–72. ISSN: 0362-1340. DOI: 10.1145/2544174.2500577. URL: http://doi.acm.org/10.1145/2544174.2500577 (cit. on p. 17).

[Bir09] Richard Bird. “Thinking Functionally with Haskell”. In: (2009). DOI: 10.1017/cbo9781316092415. URL: http://dx.doi.org/10.1017/cbo9781316092415 (cit. on p. 109).

[BL16] Patrick Baillot and Ugo Dal Lago. “Higher-order interpretations and program complexity”. In: *Inf. Comput.* 248 (2016), pp. 56–81. DOI: 10.1016/j.ic.2015.12.008. URL: https://doi.org/10.1016/j.ic.2015.12.008 (cit. on p. 62).

[Bla10] Michael Blaguszewski. “Implementing and Optimizing a Simple, Dependently-Typed Language”. MA thesis. Chalmers University of Technology, 2010. URL: http://publications.lib.chalmers.se/records/fulltext/124826.pdf (cit. on p. 16).

[BMM03] Edwin Brady, Conor McBride, and James McKinna. “Inductive Families Need Not Store Their Indices”. In: *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*. 2003, pp. 115–129. DOI: 10.1007/978-3-540-24849-1_8. URL: https://doi.org/10.1007/978-3-540-24849-1%5C_8 (cit. on p. 16).

[BP10] Eduardo Brito and Jorge Sousa Pinto. “Program Verification in SPARK and ACSL: A Comparative Case Study”. In: *Reliable Software Technologiey - Ada-Europe 2010, 15th Ada-Europe International Conference on Reliable Software Technologies, Valencia, Spain, June 14-18, 2010. Proceedings*. 2010, pp. 97–110. DOI: 10.1007/978-3-642-13550-7_7. URL: https://doi.org/10.1007/978-3-642-13550-7%5C_7 (cit. on p. 18).

[Bra] Edwin Brady. *Lectures on Implementing Idris*. URL: https://www.idris-lang.org/dependently-typed-functional-programming-with-idris-course-videos-and-slides/ (visited on 10/19/2018) (cit. on p. 16).

[Bra05] Edwin Brady. “Practical implementation of a dependently typed functional programming language”. PhD thesis. Durham University, UK, 2005. URL: http://etheses.dur.ac.uk/2800/ (cit. on p. 16).

[Car] Luca Cardelli. *A polymorphic λ-calculus with Type:Type*. URL: http://lucacardelli.name/Papers/TypeType.A4.pdf (visited on 10/19/2018) (cit. on p. 16).

[Car+11] Jacques Carette et al. *The MathScheme Library: Some Preliminary Experiments*. 2011. arXiv: `1106.1862v1 [cs.MS]` (cit. on p. 94).

[Cas+15] Pablo F. Castro et al. “Categorical foundations for structured specifications in Z”. In: *Formal Asp. Comput.* 27.5-6 (2015), pp. 831–865. DOI: `10.1007/s00165-015-0336-0`. URL: `https://doi.org/10.1007/s00165-015-0336-0` (cit. on p. 122).

[CD18] Jesper Cockx and Dominique Devriese. “Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory”. In: *J. Funct. Program.* 28 (2018), e12. DOI: `10.1017/S095679681800014X`. URL: `https://doi.org/10.1017/S095679681800014X` (cit. on p. 16).

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. “Pattern matching without K”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. 2014, pp. 257–268. DOI: `10.1145/2628136.2628139`. URL: `http://doi.acm.org/10.1145/2628136.2628139` (cit. on p. 16).

[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. 60, 75, 83, 94).

[Coh90] Edward Cohen. *Programming in the 1990s - An Introduction to the Calculation of Programs*. Texts and Monographs in Computer Science. Springer, 1990. ISBN: 978-0-387-97382-1. DOI: `10.1007/978-1-4613-9706-9`. URL: `https://doi.org/10.1007/978-1-4613-9706-9` (cit. on p. 12).

[Dia08] Razvan Diaconescu. *Institution-independent Model Theory*. 1st. Birkh&#228;user Basel, 2008. ISBN: 3764387076, 9783764387075 (cit. on p. 122).

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: `http://www.worldcat.org/oclc/01958445` (cit. on p. 12).

[DJH] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. “A formal specification of the Haskell 98 module system”. In: pp. 17–28. URL: `http://doi.acm.org/10.1145/581690.581692` (cit. on p. 18).

[Dow93] Gilles Dowek. “The Undecidability of Typability in the Lambda-Pi-Calculus”. In: *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA ’93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*. 1993, pp. 139–145. DOI: `10.1007/BFb0037103`. URL: `https://doi.org/10.1007/BFb0037103` (cit. on p. 15).

[DP15] Catherine Dubois and François Pessaux. “Termination Proofs for Recursive Functions in FoCaLiZe”. In: *Trends in Functional Programming - 16th International Symposium, TFP 2015, Sophia Antipolis, France, June 3-5, 2015. Revised Selected Papers*. 2015, pp. 136–156. DOI: `10.1007/978-3-319-39110-6\_8`. URL: `https://doi.org/10.1007/978-3-319-39110-6%5C_8` (cit. on p. 58).

[Far93] *Theory Interpretation in Simple Type Theory*. Theory interpretations formalise folklore of subtheories inheriting properties from parent theories such as satisfiability and consistency. The idea of interpreting a theory into itself is commonly done in the RATH-Agda project, for example, to obtain dual results such as those for lattices and other categorical structures. Springer-Verlag, Sept. 1993. ISBN: 3-540-58233-9. URL: http://imps.mcmaster.ca/doc/interpretations.pdf (cit. on pp. 61, 62).

[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 pp. 57, 61, 92, 122, 125).

[FM93] José Luiz Fiadeiro and T. S. E. Maibaum. "Generalising Interpretations between Theories in the context of (pi-) Institutions". In: *Theory and Formal Methods 1993, Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods, Isle of Thorns Conference Centre, Chelwood Gate, Sussex, UK, 29-31 March 1993*. 1993, pp. 126–147 (cit. on pp. 62, 122).

[Gar+09] 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 (cit. on p. 107).

[GCS14] Jason Gross, Adam Chlipala, and David I. Spivak. *Experience Implementing a Performant Category-Theory Library in Coq*. 2014. arXiv: 1401.7694v2 [math.CT] (cit. on p. 105).

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. "Eliminating Dependent Pattern Matching". In: *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*. 2006, pp. 521–540. DOI: 10.1007/11780274_27. URL: https://doi.org/10.1007/11780274%5C_27 (cit. on p. 16).

[Gon+13] Georges Gonthier et al. "How to make ad hoc proof automation less ad hoc". In: *J. Funct. Program.* 23.4 (2013), pp. 357–401. DOI: 10.1017/S0956796813000051. URL: https://doi.org/10.1017/S0956796813000051 (cit. on p. 58).

[Gra95] Paul Graham. *ANSI Common Lisp*. USA: Prentice Hall Press, 1995. ISBN: 0133708756 (cit. on p. 117).

[Gri81] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer, 1981. ISBN: 978-0-387-96480-5. DOI: 10.1007/978-1-4612-5983-1. URL: https://doi.org/10.1007/978-1-4612-5983-1 (cit. on p. 12).

[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. 80).

[Hal+] Thomas Hallgren et al. “An Overview of the Programatica Toolset”. In: *HCSS '04*. URL: http://www.cse.ogi.edu/PacSoft/projects/programatica/ (cit. on p. 18).

[Has15] Philipp Haselwarter. “Towards a Proof-Irrelevant Calculus of Inductive Constructions”. MA thesis. 2015. URL: http://www.haselwarter.org/~philipp/piCoq.pdf (cit. on p. 16).

[HS94] Martin Hofmann and Thomas Streicher. “The Groupoid Model Refutes Uniqueness of Identity Proofs”. In: *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. 1994, pp. 208–212. DOI: 10.1109/LICS.1994.316071. URL: https://doi.org/10.1109/LICS.1994.316071 (cit. on p. 16).

[Hud+07] Paul Hudak et al. “A history of Haskell: being lazy with class”. In: *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. Ed. by Barbara G. Ryder and Brent Hailpern. ACM, 2007, pp. 1–55. DOI: 10.1145/1238844.1238856. URL: https://doi.org/10.1145/1238844.1238856 (cit. on p. 109).

[KG13] Hsiang-Shang Ko and Jeremy Gibbons. “Relational Algebraic Ornaments”. In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*. DTP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 37–48. ISBN: 978-1-4503-2384-0. DOI: 10.1145/2502409.2502413. URL: http://doi.acm.org/10.1145/2502409.2502413 (cit. on p. 17).

[Kil+14] Scott Kilpatrick et al. “Backpack: retrofitting Haskell with interfaces”. In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 2014, pp. 19–32. DOI: 10.1145/2535838.2535884. URL: https://doi.org/10.1145/2535838.2535884 (cit. on p. 53).

[KS01] Wolfram Kahl and Jan Scheffczyk. “Named Instances for Haskell Type Classes”. In: 2001 (cit. on p. 21).

[Ler00] Xavier Leroy. “A modular module system”. In: *J. Funct. Program.* 10.3 (2000), pp. 269–303. DOI: 10.1017/S0956796800003683 (cit. on p. 18).

[Lip92] James Lipton. “Kripke semantics for dependent type theory and realizability interpretations”. In: *Constructivity in Computer Science*. Ed. by J. Paul Myers and Michael J. O'Donnell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 22–32. ISBN: 978-3-540-47265-0 (cit. on p. 62).

[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. 14).

[LMS10] Andres Löh, Conor McBride, and Wouter Swierstra. "A Tutorial Implementation of a Dependently Typed Lambda Calculus". In: *Fundam. Inform.* 102.2 (2010), pp. 177–207. DOI: 10.3233/FI-2010-304. URL: https://doi.org/10.3233/FI-2010-304 (cit. on p. 16).

[Luo90] Zhaohui Luo. "An extended calculus of constructions". PhD thesis. University of Edinburgh, UK, 1990. URL: http://hdl.handle.net/1842/12487 (cit. on p. 16).

[Mai05] Maria Emilia Maietti. "Modular correspondence between dependent type theories and categories including pretopoi and topoi". In: *Mathematical Structures in Computer Science* 15.6 (2005), pp. 1089–1149. DOI: 10.1017/S0960129505004962. URL: https://doi.org/10.1017/S0960129505004962 (cit. on p. 122).

[Mar+16] Simon Marlow et al. "Desugaring Haskell's do-notation into applicative operations". In: *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. Ed. by Geoffrey Mainland. ACM, 2016, pp. 92–104. ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976007. URL: https://doi.org/10.1145/2976002.2976007 (cit. on p. 107).

[Mar85] P. Martin-Löf. "Constructive Mathematics and Computer Programming". In: *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*. London, United Kingdom: Prentice-Hall, Inc., 1985, pp. 167–184. ISBN: 0-13-561465-1. URL: http://dl.acm.org/citation.cfm?id=3721.3731 (cit. on p. 16).

[McB] Conor McBride. "Ornamental Algebras, Algebraic Ornaments". In: *Unpublished Draft* (). URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/Ornament.pdf (visited on 10/19/2018) (cit. on p. 17).

[McB00a] Conor McBride. "Dependently typed functional programs and their proofs". PhD thesis. University of Edinburgh, UK, 2000. URL: http://hdl.handle.net/1842/374 (cit. on pp. 16, 155).

[McB00b] Conor McBride. "Elimination with a Motive". In: *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*. 2000, pp. 197–216. DOI: 10.1007/3-540-45842-5_13. URL: https://doi.org/10.1007/3-540-45842-5%5C_13 (cit. on p. 16).

[McK06] James McKinna. "Why dependent types matter". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 2006, p. 1. DOI: 10.1145/1111037.1111038. URL: http://doi.acm.org/10.1145/1111037.1111038 (cit. on pp. 15, 155).

[Mog91a] Eugenio Moggi. "A Cateogry-Theoretic Account of Program Modules". In: *Mathematical Structures in Computer Science* 1.1 (1991), pp. 103–139. DOI: 10.1017/S0960129500000074. URL: https://doi.org/10.1017/S0960129500000074 (cit. on p. 122).

[Mog91b] Eugenio Moggi. “Notions of Computation and Monads”. In: *Inf. Comput.* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4 (cit. on p. 107).

[MRK18] Dennis Müller, Florian Rabe, and Michael Kohlhase. “Theories as Types”. In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings.* 2018, pp. 575–590. DOI: 10.1007/978-3-319-94205-6_38. URL: https://doi.org/10.1007/978-3-319-94205-6%5C_38 (cit. on pp. 58–60).

[MS08] Nathan Mishra-Linger and Tim Sheard. “Erasure and Polymorphism in Pure Type Systems”. In: *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings.* 2008, pp. 350–364. DOI: 10.1007/978-3-540-78499-9_25. URL: https://doi.org/10.1007/978-3-540-78499-9%5C_25 (cit. on p. 16).

[MS84] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, 1984. URL: https://books.google.ca/books?id=%5C_D0ZAQAAIAAJ (cit. on p. 16).

[MT13] Assia Mahboubi and Enrico Tassi. “Canonical Structures for the working Coq user”. In: *ITP 2013, 4th Conference on Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin, and David Pichardie. Vol. 7998. LNCS. Rennes, France: Springer, July 2013, pp. 19–34. DOI: 10.1007/978-3-642-39634-2_5. URL: https://hal.inria.fr/hal-00816703 (cit. on pp. 58, 112).

[Per17] Natalie Perna. *(Re-)Creating sharing in Agda’s GHC backend.* Jan. 2017. URL: https://macsphere.mcmaster.ca/handle/11375/22177 (cit. on p. 122).

[PS90] Erik Palmgren and Viggo Stoltenberg-Hansen. “Domain Interpretations of Martin-Löf’s Partial Type Theory”. In: *Ann. Pure Appl. Logic* 48.2 (1990), pp. 135–196. DOI: 10.1016/0168-0072(90)90044-3. URL: https://doi.org/10.1016/0168-0072(90)90044-3 (cit. on p. 62).

[RS09] Florian Rabe and Carsten Schürmann. “A practical module system for LF”. In: *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP ’09, McGill University, Montreal, Canada, August 2, 2009.* 2009, pp. 40–48. DOI: 10.1145/1577824.1577831. URL: http://doi.acm.org/10.1145/1577824.1577831 (cit. on p. 52).

[Rus] Bertrand Russell. *Mathematical Logic as Based on the Theory of Types.* URL: https://fi.ort.edu.uy/innovaportal/file/20124/1/37-russell1905.pdf (visited on 10/19/2018) (cit. on p. 16).

[She] Tim Sheard. “Generic Unification via Two-Level Types and Parameterized Modules”. In: *ICFP 2001.* to appear. acm press (cit. on p. 18).

[SHH01] Tim Sheard, William Harrison, and James Hook. “Modeling the Fine Control of Demand in Haskell.” (submitted to Haskell workshop 2001). 2001 (cit. on p. 18).

[Shu16] Michael Shulman. *Categorical logic from a categorical point of view.* 2016. URL: https://mikeshulman.github.io/catlog/catlog.pdf (visited on 10/19/2018) (cit. on p. 122).

[Str93] Thomas Streicher. "Investigations Into Intensional Type Theory". PhD thesis. 1993. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf (cit. on p. 16).

[Swi08] Wouter Swierstra. "Data types à la carte". In: *J. Funct. Program.* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758. URL: https://doi.org/10.1017/S0956796808006758 (cit. on p. 112).

[TB] Matus Tejiscak and Edwin Brady. "Practical Erasure in Dependently Typed Languages". In: *Unpublished Draft* (). URL: https://eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf (visited on 10/19/2018) (cit. on p. 16).

[VME18] Grigoriy Volkov, Mikhail U. Mandrykin, and Denis Efremov. "Lemma Functions for Frama-C: C Programs as Proofs". In: *CoRR* abs/1811.05879 (2018). arXiv: 1811.05879. URL: http://arxiv.org/abs/1811.05879 (cit. on p. 18).

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* USA: Prentice-Hall, Inc., 1996. ISBN: 0139484728 (cit. on p. 137).

[Wei] Stephanie Weirich. *2014 OPLSS Lectures* Designing Dependently-Typed Programming Languages. URL: https://www.cs.uoregon.edu/research/summerschool/summer14/curriculum.html (visited on 10/19/2018) (cit. on p. 16).

[Wer08] Benjamin Werner. "On the Strength of Proof-irrelevant Type Theories". In: *Logical Methods in Computer Science* 4.3 (2008). DOI: 10.2168/LMCS-4(3:13)2008. URL: https://doi.org/10.2168/LMCS-4(3:13)2008 (cit. on p. 16).

[WK18] Philip Wadler and Wen Kokke. *Programming Language Foundations in Agda.* 2018. URL: https://plfa.github.io/ (visited on 10/12/2018) (cit. on p. 155).

Appendix A

Context Implementation

Below is the entirety of the `Context` library.

The Context Library

```
-- Agda version 2.6.0.1
-- Standard library version 1.2

module Context where
```

Also included are unit tests, evidence for claims made in the thesis proper, and a brief case-study on graphs to demonstrate some features of the Context library that are necessary for practical use, such as field projections, but which did not receive attention in the paper proper.

A.1 Imports

The Context Library

```
open import Level renaming (_⊔_ to _⊎_; suc to ℓsuc; zero to ℓ₀)
open import Relation.Binary.PropositionalEquality
open import Relation.Nullary

open import Data.Nat
open import Data.Fin  as Fin using (Fin)
open import Data.Maybe  hiding (_>>=_)

open import Data.Bool using (Bool ; true ; false)
open import Data.List as List using (List ; [] ; _::_ ; _::ʳ_; sum)

ℓ₁   = Level.suc ℓ₀
```

A.2 Quantifiers Π:●/Σ:● and Products/Sums

We shall using Z-style quantifier notation Woodcock and Davies [WD96] in which the quantifier dummy variables are separated from the body by a large bullet.

In Agda, we use `\:` to obtain the "ghost colon" since standard colon `:` is an Agda operator.

Even though Agda provides `∀ (x : τ) → fx` as a built-in syntax for Π-types, we have chosen the Z-style one below to mirror the notation for Σ-types, which Agda provides as `record` declarations. In the paper proper, in the definition of bind, the subtle shift between Σ-types and Π-types is easier to notice when the notations are so similar that only the quantifier symbol changes.

The Context Library

```
open import Data.Empty using (⊥)
open import Data.Sum
open import Data.Product
open import Function using (_∘_)

Σ:• : ∀ {a b} (A : Set a) (B : A → Set b) → Set _
Σ:• = Σ

infix -666 Σ:•
syntax Σ:• A (λ x → B) = Σ x : A • B

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

infix -666 Π:•
syntax Π:• A (λ x → B) = Π x : A • B

record ⊤ {ℓ} : Set ℓ where
  constructor tt

𝟙 = ⊤ {ℓ₀}
𝟘 = ⊥
```

A.3 Reflection

We form a few metaprogramming utilities we would have expected to be in the standard library.

The Context Library

```
import Data.Unit as Unit
open import Reflection hiding (name; Type) renaming (_>>=_ to _>>=ₘ_)
```

Before continuing, there are a few difficulties about Agda's metaprogramming capabilities that should be mentioned:

1. Even when recursion is on structurally smaller terms of abstract syntax trees, termination cannot be automatically deduced. As such, we request Agda to believe us that certain definitions are terminating.

2. Since Agda macros cannot be recursive —possibly due to issues of termination— an idiom we use to define a recursive operation on terms then wrap that in Agda's type-checking monad to form macros.

3. Sometimes, no matter how explicit we make certain affairs, macro invocations will complain about being unable to infer certain details. As a workaround, we type any declaration involving a macro invocation before using it —inference is difficult in dependently-typed settings and even worse in the presence of metaprogramming.

A.3.1 Single argument application

The Context Library

```
_app_ : Term → Term → Term
(def f args) app arg' = def f (args ::ʳ arg (arg-info visible relevant) arg')
(con f args) app arg' = con f (args ::ʳ arg (arg-info visible relevant) arg')
{-# CATCHALL #-}
tm app arg' = tm
```

Notice that we maintain existing applications:

```
quoteTerm (f x) app quoteTerm y  ≈  quoteTerm (f x y)
```

A.3.2 Reify ℕ term encodings as ℕ values

The Context Library

```
toℕ : Term → ℕ
toℕ (lit (nat n)) = n
{-# CATCHALL #-}
toℕ _ = 0
```

A.3.3 The Length of a Term

The Context Library

```
arg-term : ∀ {ℓ} {A : Set ℓ} → (Term → A) → Arg Term → A
arg-term f (arg i x) = f x

{-# TERMINATING #-}
lengthₜ : Term → ℕ
lengthₜ (var x args)      = 1 + sum (List.map (arg-term lengthₜ ) args)
lengthₜ (con c args)      = 1 + sum (List.map (arg-term lengthₜ ) args)
lengthₜ (def f args)      = 1 + sum (List.map (arg-term lengthₜ ) args)
lengthₜ (lam v (abs s x)) = 1 + lengthₜ x
lengthₜ (pat-lam cs args) = 1 + sum (List.map (arg-term lengthₜ ) args)
lengthₜ (Π[ x : A ] Bx)   = 1 + lengthₜ Bx
{-# CATCHALL #-}
-- sort, lit, meta, unknown
lengthₜ t = 0
```

Here is an example use:

The Context Library

```
_ : lengthₜ (quoteTerm (Σ x : ℕ • x ≡ x)) ≡ 10
_ = refl
```

A.3.4 Decreasing de Brujin Indices

Given a quantification (⊕ `x` : τ • `fx`), its body `fx` may refer to a free variable `x`. If we decrement all de Bruijn indices `fx` contains, then there would be no reference to `x`.

The Context Library

```
var-dec₀ : (fuel : ℕ) → Term → Term
var-dec₀ zero t  = t
-- Let’s use an “impossible” term.
var-dec₀ (suc n) (var zero args)      = def (quote ⊥) []
var-dec₀ (suc n) (var (suc x) args)   = var x args
var-dec₀ (suc n) (con c args)         = con c (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (def f args)         = def f (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (lam v (abs s x))    = lam v (abs s (var-dec₀ n x))
var-dec₀ (suc n) (pat-lam cs args)    = pat-lam cs (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (Π[ s : arg i A ] B) = Π[ s : arg i (var-dec₀ n A) ] var-dec₀ n B
{-# CATCHALL #-}
-- sort, lit, meta, unknown
var-dec₀ n t = t
```

In the paper proper, `var-dec` was mentioned once under the name ⇊.

The Context Library

```
var-dec : Term → Term
var-dec t = var-dec₀ (lengthₜ t) t
```

Notice that we made the decision that `x`, the body of `(⊕ x • x)`, will reduce to **0**, the empty type. Indeed, in such a situation the only Debrujin index cannot be reduced further. Here is an example:

The Context Library

```
_ : ∀ {x : ℕ} → var-dec (quoteTerm x) ≡ quoteTerm ⊥
_ = refl
```

A.4 Context Monad

The Context Library

```
Context = λ ℓ → ℕ → Set ℓ

infix -1000 ‘_
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S

End : ∀ {ℓ} → Context ℓ
End = ‘ ⊤

End₀ = End {ℓ₀}

_>>=_ : ∀ {a b}
      → (Γ : Set a)   -- Main diference
      → (Γ → Context b)
      → Context (a ⊎ b)
(Γ >>= f) ℕ.zero   = Σ γ : Γ • f γ 0
(Γ >>= f) (suc n) = (γ : Γ) → f γ n
```

A.5 ⟨⟩ Notation

The Context Library

```
-- Expressions of the form "··· , tt" may now be written "⟨ ··· ⟩"
infixr 5 ⟨ _⟩
⟨⟩ : ∀ {ℓ} → ⊤ {ℓ}
⟨⟩ = tt

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

_⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × ⊤ {ℓ}
s ⟩ = s , tt
```

A.6 DynamicSystem Context

The Context Library

```
DynamicSystem : Context (ℓsuc Level.zero)
DynamicSystem = do X ← Set
                   z ← X
                   s ← (X → X)
                   End {Level.zero}

-- Records with n-Parameters, n : 0..3
A B C D : Set₁
A = DynamicSystem 0 -- Σ X : Set  • Σ z : X  • Σ s : X → X  • ⊤
B = DynamicSystem 1 --  (X : Set) → Σ z : X  • Σ s : X → X  • ⊤
C = DynamicSystem 2 --  (X : Set)    (z : X) → Σ s : X → X  • ⊤
D = DynamicSystem 3 --  (X : Set)    (z : X) →  (s : X → X) → ⊤

_ : A ≡ (Σ X : Set  • Σ z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : B ≡ (Π X : Set  • Σ z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : C ≡ (Π X : Set  • Π z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : D ≡ (Π X : Set  • Π z : X  • Π s : (X → X)  • ⊤) ; _ = refl

stability : ∀ {n} →   DynamicSystem (3 + n)
                   ≡ DynamicSystem  3
stability = refl

B-is-empty : ¬ B
B-is-empty b = proj₁( b ⊥)

𝒩₀ : DynamicSystem 0
𝒩₀ = ℕ , 0 , suc , tt

𝒩 : DynamicSystem 0
𝒩 = ⟨ ℕ , 0 , suc ⟩

B-on-ℕ : Set
B-on-ℕ = let X = ℕ in Σ z : X  • Σ s : (X → X)  • ⊤

ex : B-on-ℕ
ex = ⟨ 0 , suc ⟩
```

A.7 $\Pi{\rightarrow}\lambda$

The Context Library

```
Π→λ-helper : Term → Term
Π→λ-helper (pi  a b)          = lam visible b
Π→λ-helper (lam a (abs x y)) = lam a (abs x (Π→λ-helper y))
{-# CATCHALL #-}
Π→λ-helper x = x

macro
  Π→λ : Term → Term → TC Unit.⊤
  Π→λ tm goal = normalise tm >>=ₘ λ tm' → unify (Π→λ-helper tm') goal
```

A.8 $\mathtt{id}_{i+1} \approx \Pi{\rightarrow}\lambda\ \mathtt{id}_i$

The Context Library

```
_ : id₁ ≡ Π→λ id₀
_ = refl

_ : id₂ ≡ Π→λ id₁
_ = refl
```

A.9 _:waist_

The Context Library

```
waist-helper : ℕ → Term → Term
waist-helper zero t    = t
waist-helper (suc n) t = waist-helper n (Π→λ-helper t)

macro
  _:waist_ : Term → Term → Term → TC Unit.⊤
  _:waist_ t n goal =      normalise (t app n)
                     >>=ₘ λ t' → unify (waist-helper (toℕ n) t') goal
```

A.10 DynamicSystem :waist i

The Context Library

```
A' : Set₁
B' : ∀ (X : Set) → Set
C' : ∀ (X : Set) (x : X) → Set
D' : ∀ (X : Set) (x : X) (s : X → X) → Set

A' = DynamicSystem :waist 0
B' = DynamicSystem :waist 1
C' = DynamicSystem :waist 2
D' = DynamicSystem :waist 3

𝒩⁰ : A'
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : B' ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : C' ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : D' ℕ 0 suc
𝒩³ = ⟨⟩
```

It may be the case that Γ `0` ≡ Γ `:waist 0` for every context Γ.

The Context Library

```
_ : DynamicSystem 0 ≡ DynamicSystem :waist 0
_ = refl
```

A.11 Field projections

The Context Library

```
Field₀ : ℕ → Term → Term
Field₀ zero c    = def (quote proj₁) (arg (arg-info visible relevant) c :: [])
Field₀ (suc n) c = Field₀ n (def (quote proj₂) (arg (arg-info visible relevant) c ::
 ↪  []))

macro
  Field : ℕ → Term → Term → TC Unit.⊤
  Field n t goal = unify goal (Field₀ n t)
```

An example usage can be found below in the setting of graphs.

A.12 Termtypes

Using the guiding calculation outlined in the paper proper we shall form $\mathtt{D}_i$ for each stage in the calculation.

A.12.1 Stage 1: Records

The Context Library

```
D₁ = DynamicSystem 0

1-records : D₁ ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • ⊤)
1-records = refl
```

A.12.2 Stage 2: Parameterised Records

The Context Library

```
D₂ = DynamicSystem :waist 1

2-funcs : D₂ ≡ (λ (X : Set) → Σ z : X • Σ s : (X → X) • ⊤)
2-funcs = refl
```

A.12.3 Stage 3: Sources

Let's begin with an example to motivate the definition of `sources`.

The Context Library

```
_ :   quoteTerm (∀ {x : ℕ} → ℕ)
    ≡ pi (arg (arg-info hidden relevant) (quoteTerm ℕ)) (abs "x" (quoteTerm ℕ))
_ = refl
```

We now form two sources-helper utilities, although we suspect they could be combined into one function.

The Context Library

```
sources₀ : Term → Term
-- Otherwise:
sources₀ (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) =
    def (quote _×_) (vArg A
                    :: vArg (def (quote _×_)
                                (vArg (var-dec Ba)
                                    :: vArg (var-dec (var-dec (sources₀ Cab))) ::
 ↪  []))
                    :: [])
sources₀ (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 𝟘
sources₀ (Π[ x : arg i A ] Bx) = A
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources₀ t = quoteTerm 𝟙

{-# TERMINATING #-}
sources₁ : Term → Term
sources₁ (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 𝟘
sources₁ (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) = def (quote _×_) (vArg A ::
  vArg (def (quote _×_) (vArg (var-dec Ba)
                              :: vArg (var-dec (var-dec (sources₀ Cab))) :: [])) ::
 ↪  [])
sources₁ (Π[ x : arg i A ] Bx) = A
sources₁ (def (quote Σ) (ℓ₁ :: ℓ₂ :: τ :: body))
    = def (quote Σ) (ℓ₁ :: ℓ₂ :: map-Arg sources₀ τ :: List.map (map-Arg sources₁)
 ↪  body)
-- This function introduces 𝟙s, so let's drop any old occurances a la 𝟘.
sources₁ (def (quote ⊤) _) = def (quote 𝟘) []
sources₁ (lam v (abs s x))     = lam v (abs s (sources₁ x))
sources₁ (var x args) = var x (List.map (map-Arg sources₁) args)
sources₁ (con c args) = con c (List.map (map-Arg sources₁) args)
sources₁ (def f args) = def f (List.map (map-Arg sources₁) args)
sources₁ (pat-lam cs args) = pat-lam cs (List.map (map-Arg sources₁) args)
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources₁ t = t
```

We now form the macro and some unit tests.

The Context Library

```
macro
  sources : Term → Term → TC Unit.⊤
  sources tm goal = normalise tm >>=ₘ λ tm’ → unify (sources₁ tm’) goal

_ : sources (ℕ → Set) ≡ ℕ
_ = refl

_ : sources (Σ x : (ℕ → Fin 3) • ℕ) ≡ (Σ x : ℕ • ℕ)
_ = refl

_ : ∀ {ℓ : Level} {A B C : Set}
  → sources (Σ x : (A → B) • C) ≡ (Σ x : A • C)
_ = refl

_ : sources (Fin 1 → Fin 2 → Fin 3) ≡ (Σ _ : Fin 1 • Fin 2 × 𝟙)
_ = refl

_ : sources (Σ f : (Fin 1 → Fin 2 → Fin 3 → Fin 4) • Fin 5)
  ≡ (Σ f : (Fin 1 × Fin 2 × Fin 3) • Fin 5)
_ = refl

_ : ∀ {A B C : Set} → sources (A → B → C) ≡ (A × B × 𝟙)
_ = refl

_ : ∀ {A B C D E : Set} → sources (A → B → C → D → E)
                        ≡ Σ A (λ _ → Σ B (λ _ → Σ C (λ _ → Σ D (λ _ → ⊤))))
_ = refl
```

Design decision: Types starting with implicit arguments are *invariants*, not *constructors*.

The Context Library

```
-- one implicit
_ : sources (∀ {x : ℕ} → x ≡ x) ≡ 𝟘
_ = refl

-- multiple implicits
_ : sources (∀ {x y z : ℕ} → x ≡ y) ≡ 𝟘
_ = refl
```

The third stage can now be formed.

The Context Library

```
D₃ = sources D₂

3-sources : D₃ ≡ λ (X : Set) → Σ z : 𝟙 • Σ s : X • 𝟘
3-sources = refl
```

A.12.4 Stage 4: Σ→⊎ –Replacing Products with Sums

The Context Library

```
{-# TERMINATING #-}
Σ→⊎₀ : Term → Term
Σ→⊎₀ (def (quote Σ) (h₁ :: h₀ :: arg i A :: arg i₁ (lam v (abs s x)) :: []))
  =  def (quote _⊎_) (h₁ :: h₀ :: arg i A :: vArg (Σ→⊎₀ (var-dec x)) :: [])
-- Interpret “End” in do-notation to be an empty, impossible, constructor.
Σ→⊎₀ (def (quote ⊤) _) = def (quote ⊥) []
 -- Walk under λ’s and Π’s.
Σ→⊎₀ (lam v (abs s x)) = lam v (abs s (Σ→⊎₀ x))
Σ→⊎₀ (Π[ x : A ] Bx) = Π[ x : A ] Σ→⊎₀ Bx
{-# CATCHALL #-}
Σ→⊎₀ t = t

macro
  Σ→⊎ : Term → Term → TC Unit.⊤
  Σ→⊎ tm goal = normalise tm >>=ₘ λ tm’ → unify (Σ→⊎₀ tm’) goal
```

Unit tests:

The Context Library

```
_ : Σ→⊎ (Π X : Set • (X → X))       ≡ (Π X : Set • (X → X)); _ = refl
_ : Σ→⊎ (Π X : Set • Σ s : X • X) ≡ (Π X : Set • X ⊎ X)  ; _ = refl
_ : Σ→⊎ (Π X : Set • Σ s : (X → X) • X) ≡ (Π X : Set • (X → X) ⊎ X)  ; _ = refl
_ : Σ→⊎ (Π X : Set • Σ z : X • Σ s : (X → X) • ⊤ {ℓ₀}) ≡ (Π X : Set • X ⊎ (X →
 ↪  X) ⊎ ⊥)
_ = refl
```

The Context Library

```
D₄ = Σ→⊎ D₃

4-unions : D₄ ≡ λ X → 𝟙 ⊎ X ⊎ 𝟘
4-unions = refl
```

A.12.5 Stage 5: Fixpoint and proof that $\mathbb{D} \cong \mathbb{N}$

Since we want to define algebraic data-types as fixed-points, we are led inexorably to using a recursive type that fails to be positive.

The Context Library

```
{-# NO_POSITIVITY_CHECK #-}
data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
  μ : F (Fix F) → Fix F
```

The Context Library

```
module termtype[DynamicSystem]≅ℕ where

  𝔻 = Fix D₄

  -- Pattern synonyms for more compact presentation
  pattern zeroD   = μ (inj₁ tt)        -- : 𝔻
  pattern sucD e = μ (inj₂ (inj₁ e)) -- : 𝔻 → 𝔻

  to : 𝔻 → ℕ
  to zeroD    = 0
  to (sucD x) = suc (to x)

  from : ℕ → 𝔻
  from zero     = zeroD
  from (suc n) = sucD (from n)

  to∘from : ∀ n → to (from n) ≡ n
  to∘from zero     = refl
  to∘from (suc n) = cong suc (to∘from n)

  from∘to : ∀ d → from (to d) ≡ d
  from∘to zeroD     = refl
  from∘to (sucD x) = cong sucD (from∘to x)
```

A.12.6 `termtype` and `Inj` macros

We summarise the stages together into one macro: "`termtype : UnaryFunctor → Type`".

The Context Library

```
macro
  termtype : Term → Term → TC Unit.⊤
  termtype tm goal =
              normalise tm
         >>=ₘ λ tm' → unify goal (def (quote Fix) ((vArg (Σ→⊎₀ (sources₁ tm')))
 ↪  :: []))
```

It is interesting to note that in place of `pattern` clauses, say for languages that do not support them, we would resort to "fancy injections".

The Context Library

```
Inj₀ : ℕ → Term → Term
Inj₀ zero c    = con (quote inj₁) (arg (arg-info visible relevant) c :: [])
Inj₀ (suc n) c = con (quote inj₂) (vArg (Inj₀ n c) :: [])

-- Duality!
-- i-th projection: proj₁ ∘ (proj₂ ∘ ⋯ ∘ proj₂)
-- i-th injection:  (inj₂ ∘ ⋯ ∘ inj₂) ∘ inj₁

macro
  Inj : ℕ → Term → Term → TC Unit.⊤
  Inj n t goal = unify goal ((con (quote μ) []) app (Inj₀ n t))
```

With this alternative, we regain the “user chosen constructor names” for 𝔻:

The Context Library

```
startD : 𝔻
startD = Inj 0 (tt {ℓ₀})

nextD' : 𝔻 → 𝔻
nextD' d = Inj 1 d
```

A.13 The _:kind_ meta-primitive

The Context Library

```
data Kind : Set where
  ‘record    : Kind
  ‘typeclass : Kind
  ‘data      : Kind

macro
  _:kind_ : Term → Term → Term → TC Unit.⊤
  _:kind_ t (con (quote ‘record) _)    goal = normalise (t app (quoteTerm 0))
                      >>=ₘ λ t' → unify (waist-helper 0 t') goal
  _:kind_ t (con (quote ‘typeclass) _) goal = normalise (t app (quoteTerm 1))
                      >>=ₘ λ t' → unify (waist-helper 1 t') goal
  _:kind_ t (con (quote ‘data) _) goal = normalise (t app (quoteTerm 1))
                      >>=ₘ λ t' → normalise (waist-helper 1 t')
                      >>=ₘ λ t'' → unify goal (def (quote Fix)
                                                 ((vArg (Σ→⊎₀ (sources₁ t''))) ::
 ↪  []))
  _:kind_ t _ goal = unify t goal
```

Informally, `_:kind_` behaves as follows:

```
                                              The Context Library

𝒞 :kind ‘record    = 𝒞 :waist 0
𝒞 :kind ‘typeclass = 𝒞 :waist 1
𝒞 :kind ‘data      = termtype (𝒞 :waist 1)
```

A.14 Example: Graphs in Two Ways

There are two ways to implement the type of graphs in the dependently-typed language Agda: Having the vertices be a parameter or having them be a field of the record. Then there is also the syntax for graph vertex relationships. Suppose a library designer decides to work with fully bundled graphs, `Graph`$_0$ below, then a user decides to write the function `comap`, which relabels the vertices of a graph, using a function `f` to transform vertices.

```
                                              The Context Library

record Graph₀ : Set₁ where
  constructor ⟨_,_⟩₀
  field
    Vertex : Set
    Edges : Vertex → Vertex → Set

open Graph₀

comap₀ : {A B : Set}
       → (f : A → B)
       → (Σ G : Graph₀ • Vertex G ≡ B)
       → (Σ H : Graph₀ • Vertex H ≡ A)
comap₀ {A} f (G , refl) = ⟨ A , (λ x y → Edges G (f x) (f y)) ⟩₀ , refl
```

Since the vertices are packed away as components of the records, the only way for `f` to refer to them is to awkwardly refer to seemingly arbitrary types, only then to have the vertices of the input graph `G` and the output graph `H` be constrained to match the type of the relabelling function `f`. Without the constraints, we could not even write the function for `Graph`$_0$. With such an importance, it is surprising to see that the occurrences of the constraint obligations are uninsightful `refl`-exivity proofs.

What the user would really want is to unbundle `Graph`$_0$ at will, to expose the first argument, to obtain `Graph`$_1$ below. Then, in stark contrast, the implementation `comap`$_1$ does not carry any excesses baggage at the type level nor at the implementation level.

The Context Library

```
record Graph₁ (Vertex : Set) : Set₁ where
  constructor ⟨_⟩₁
  field
    Edges : Vertex → Vertex → Set

comap₁ : {A B : Set}
       → (f : A → B)
       → Graph₁ B
       → Graph₁ A
comap₁ f ⟨ edges ⟩₁ = ⟨ (λ x y → edges (f x) (f y)) ⟩₁
```

With `Graph`$_1$, one immediately sees that the `comap` operation "pulls back" the vertex type. Such an observation for `Graph`$_0$ is not as easy; requiring familiarity with quantifier laws such as the one-point rule and quantifier distributivity.

A.15 Example: Graphs with Delayed Unbundling

The ubiquitous graph structure is contravariant in its collection of vertices. Recall that a multi-graph, or quiver, is a collection of vertices along with a collection of edges between any two vertices; here's the traditional record form:

The Context Library

```
Graph  : Context ℓ₁
Graph  = do Vertex ← Set
            Edges  ← (Vertex → Vertex → Set)
            End {ℓ₀}
```

Using the record form, it is awkward to phrase contravariance, which simply "relabels the vertices". Even worse, the awkward phrasing only serves to ensure certain constraints hold —which are reified at the value level via the uninsightful `refl`-exivity proof.

The Context Library

```
pattern ⟨_,_⟩ V E = (V , E , tt)

comap₀' : ∀ {A B : Set}
      → (f : A → B)
      → Σ G : Graph :kind ‘record • Field 0 G ≡ B
      → Σ G : Graph :kind ‘record • Field 0 G ≡ A
comap₀' {A} {B} f (⟨ .B , edgs ⟩ , refl) = (A , (λ a₁ a₂ → edgs (f a₁) (f a₂)) , tt)
 ↪  , refl
```

Without redefining graphs, we can phrase the definition at the 'typeclass' level —i.e., records parameterised by the vertices. This form is not only clearer and easier to implement at the value-level, it also makes it clear that we are "pulling back" the vertex type and so have also shown graphs are closed under reducts.

The Context Library

```
pattern ⟨_⟩¹ E = (E , tt)

-- Way better and less awkward!
comap’ : ∀ {A B : Set}
     → (f : A → B)
     → (Graph :kind ‘typeclass) B
     → (Graph :kind ‘typeclass) A
comap’ f ⟨ edgs ⟩¹ = ⟨ (λ a₁ a₂ → edgs (f a₁) (f a₂)) ⟩¹
```

Excellent, we can unbundle at will.

Appendix B

A Whirlwind Tour of Agda

Agda McKinna [McK06], McBride [McB00a], Bove and Dybjer [BD08], and Wadler and Kokke [WK18] is based on Martin-Löf's intuitionistic type theory. By identifying types with terms, the type of small types is a larger type; e.g., `ℕ : Set`$_0$ and `Set`$_i$ `: Set`$_{i+1}$ —the indices `i` are called *levels* and the small type `Set`$_0$ is abbreviated as `Set`. In some regard, Agda adds *harmonious* support for dependent types to Haskell.

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

B.1 Dependent Functions

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

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

The Identity Function

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

id₁ id₂ id₃ : (X : Set) → X → X

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

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

Inferring Arguments...

```
id : {X : Set} → X → X
id x = x

sad : ℕ
sad = id₀ ℕ 3

nice : ℕ
nice = id 3
```

...and Explicitly Passsing Implicits

```
explicit : ℕ
explicit = id {ℕ} 3

explicit’ : ℕ
explicit’ = id₀ _ 3


.
```

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

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

```
   {x : _} {y : _} (z : _) → ⋯
≈  ∀ {x y} z → ⋯
```

```
   (a₁ : A) → (a₂ : A) → (b : B) → ⋯
≈  (a₁ a₂ : A) (b : B) → ⋯
```

B.2 Dependent Datatypes

Algebraic datatypes are introduced with a `data` declaration, giving the name, arguments, and type of the datatype as well as the constructors and their types. Below we define the datatype of lists of a particular length.

Vectors —ℕ-indexed Lists

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

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

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

Safe Head

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

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

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

Catenation is a ++⟶+ Homomorphism

```
infixr 40 _++_

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

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

B.3 Propositional Equality

An example of propositions-as-types is a definition of the identity relation —the least reflexive relation. For a type `A` and an element `x` of `A`, we define the family of proofs of "being equal to x" by declaring only one inhabitant at index `x`.

Propositional Equality

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

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

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

Transport along proofs

```
subst : {A : Set} {P : A → Set} {l r : A} → l ≡ r → P l → P r
subst refl it = it
```

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

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

Examples of Propositional and Definitional Equality

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

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

B.4 Calculational Proofs —Making Use of Unicode Mixfix Lexemes

School math classes show calculations as follows.

```
  p
≡⟨ reason why p ≡ q ⟩
  q
≡⟨ reason why q ≡ r ⟩
  r
□
```

Calculational Proof Syntax Embedded As Proof Forming Functions

```
infixr 5 _≡⟨_⟩_
infix  6 _□

_□ : {A : Set} (a : A) → a ≡ a
_ □ = refl

_≡⟨_⟩_ : {A : Set} (p {q r} : A)
       → p ≡ q → q ≡ r → p ≡ r
_ ≡⟨ refl ⟩ refl = refl
```

We can treat these pieces as Agda *mixfix* identifiers and associate to the right to obtain: `p ≡⟨ reason`$_1$ `⟩ (q ≡⟨ reason`$_2$ `⟩ (r □))`. We can code this up, as show above on the right.

B.5 Modules —Namespace Management

Agda modules are not a first-class construct, yet.

◇ Within a module, we may have nested module declarations.

◇ All names in a module are public, unless declared `private`.

A Simple Module

```
module M where

  𝒩 : Set
  𝒩 = ℕ

  private
    x : ℕ
    x = 3

  y : 𝒩
  y = x + 1
```

Using It

```
use₀ : M.𝒩
use₀ = M.y

use₁ : ℕ
use₁ = y
  where open M

open M

use₂ : ℕ
use₂ = y
```

Parameterised Modules

```
module M’ (x :
 ↪  ℕ)
  where
    y : ℕ
    y = x + 1
```

Names=Functions

```
exposed : (x :
 ↪  ℕ)
        → ℕ
exposed = M’.y
```

Using Them

```
use’₀ : ℕ
use’₀ = M’.y 3

module M’’ = M’
 ↪  3

use’’ : ℕ
use’’ = M’’.y

use’₁ : ℕ
use’₁ = y
  where
    open M’ 3
```

◇ Public names may be accessed by qualification or by opening them locally or globally.

◇ Modules may be parameterised by arbitrarily many values and types —but not by other modules.

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

"Using Them":

◇ This explains how names in parameterised modules are used: They are treated as functions.

◇ We may prefer to instantiate some parameters and name the resulting module.

◇ However, we can still `open` them as usual.

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

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

Table B.1: Module combinators supported in the current implementation of Agda

Splitting a program over several files will improve type checking performance, since when you are making changes the type checker only has to check the files that are influenced by the change.

◇ `import X.Y.Z`: Use the definitions of module `Z` which lives in file `./X/Y/Z.agda`.

◇ `open M public`: Treat the contents of `M` as if they were public contents of the current module.

So much for Agda modules.

B.6 Records

A record type is declared much like a datatype where the fields are indicated by the `field` keyword. The nature of records is summarised by the following equation.

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

The class of types along with a value picked out

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

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

Defining Instances

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

ex₁ : PointedSet
ex₁ = MkIt ℕ 3

open PointedSet

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

Within the Emacs interface, start with `ex₂ = ?`, then in the hole enter `C-c C-c RET` to obtain the *co-pattern* setup. Two tuples are the same when they have the same components, likewise a record is defined by its projections, whence *co-patterns*. If you are using many local definitions, you likely want to use co-patterns.

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

Simple Uses

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

use¹ : ℕ
use¹ = point where open PointedSet ex₀

open PointedSet

use² : ℕ
use² = blind ex₀ true
```

You can even pattern match on records—they're just `data` after all!

Pattern Matching on Records

```
use³ : (P : PointedSet) → Carrier P
use³ record {Carrier = C; point = x}
  = x

use⁴ : (P : PointedSet) → Carrier P
use⁴ (MkIt C x)
  = x
```

So much for records.

B.7 Interacting with the real world —Compilation, Haskell, and IO

In order to be useful, a program must interact with the real world. Agda relegates the work to Haskell. The only concept here that is used in later sections will be Agda's Do-Notation, and so the purpose of this section is to demonstrate how to use it in a real scenario.

An Agda program module containing a `main` function is compiled into a standalone executable with `agda --compile myfile.agda`. If the module has no main file, use the flag `--no-main`. If you only want the resulting Haskell, not necessarily an executable program, then use the flag `--ghc-dont-call-ghc`.

The type of `main` should be `Agda.Builtin.IO.IO A`, for some `A`; this is just a proxy to Haskell's `IO`. We may `open import IO.Primitive` to get *this* `IO`, but this one works with costrings, which are a bit awkward. Instead, we use the standard library's wrapper type, also named `IO`. Then we use `run` to move from `IO` to `Primitive.IO`; conversely one uses `lift`.

Necessary Imports

```
open import Data.Nat                 using (ℕ; suc)
open import Data.Nat.Show            using (show)
open import Data.Char                using (Char)
open import Data.List as L           using (map; sum; upTo)
open import Function                 using (_$_; const; _∘_)
open import Data.String as S         using (String; _++_; fromList)
open import Agda.Builtin.Unit        using (⊤)
open import Codata.Musical.Colist    using (take)
open import Codata.Musical.Costring  using (Costring)
open import Data.BoundedVec.Inefficient as B using (toList)
open import Agda.Builtin.Coinduction using (SHARP_)
open import IO as IO                 using (run ; putStrLn ; IO)
import IO.Primitive as Primitive
```

Agda has ***no*** *primitives for side-effects, instead it allows arbitrary Haskell functions to be imported as axioms, whose definitions are only used at run-time.*

Agda lets us use `do`-notation as in Haskell. To do so, methods named `_»_` and `_»=_` need to be in scope —that is all. The type of `IO._»_` takes two "lazy" IO actions and yield a non-lazy IO action. The one below is a homogeneously typed version.

Non-lazy Do-combinators

```
infixr 1 _>>=_ _>>_

_>>=_ : ∀ {ℓ} {α β : Set ℓ} → IO α → (α → IO β) → IO β
this >>= f = SHARP this IO.>>= λ x → SHARP f x

_>>_ : ∀{ℓ} {α β : Set ℓ} → IO α → IO β → IO β
x >> y = x >>= const y
```

Oddly, Agda's standard library comes with `readFile` and `writeFile`, but the symmetry ends there since it provides `putStrLn` but not `getLine`. Mimicking the `IO.Primitive` module, we define *two* versions ourselves as proxies for Haskell's `getLine` —the second one below is bounded by 100 characters, whereas the first is not.

Postulating Foreign Haskell Functions

```
postulate
  getLine∞ : Primitive.IO Costring

{-# FOREIGN GHC
  toColist :: [a] -> MAlonzo.Code.Codata.Musical.Colist.AgdaColist a
  toColist []       = MAlonzo.Code.Codata.Musical.Colist.Nil
  toColist (x : xs) =
    MAlonzo.Code.Codata.Musical.Colist.Cons x (MAlonzo.RTE.Sharp (toColist xs))
#-}

{- Haskell's prelude is implicitly available; this is for demonstration. -}
{-# FOREIGN GHC import Prelude as Haskell #-}
{-# COMPILE GHC getLine∞  = fmap toColist Haskell.getLine #-}

-- (1)
-- getLine : IO Costring
-- getLine = IO.lift getLine∞

getLine : IO String
getLine = IO.lift
  $ getLine∞ Primitive.>>= (Primitive.return ∘ S.fromList ∘ B.toList ∘ take 100)
```

We obtain `MAlonzo` strings, then convert those to colists, then eventually lift those to the wrapper `IO` type.

Let's also give ourselves Haskell's `read` method.

Postulating Haskell's 'read'

```
postulate readInt  : L.List Char → ℕ
{-# COMPILE GHC readInt = \x -> read x :: Integer  #-}
```

Now we write our `main` method.

An Agda Program: Triangle Numbers with IO

```
main : Primitive.IO ⊤
main = run do putStrLn "Hello, world! I'm a compiled Agda program!"

              putStrLn "What is your name?"
              name ← getLine

              putStrLn "Please enter a number."
              num ← getLine
              let tri = show $ sum $ upTo $ suc $ readInt $ S.toList num
              putStrLn $ "The triangle number of " ++ num ++ " is " ++ tri

              putStrLn "Bye, "
              -- IO.putStrLn∞ name  {- If we use approach (1) above. -}
              putStrLn $ "\t" ++ name
```

For example, the 12th triangle number is $\sum_{i=0}^{12} i = 78$. Interestingly, when an integer parse fails, the program just crashes.

Calling this file `CompilingAgda.agda`, we may compile then run it with:

Compiling The Program

```
NAME=CompilingAgda; time agda --compile $NAME.agda; ./$NAME
```

The very first time you compile may take ∼80 seconds since some prerequisites need to be compiled, but future compilations are within ∼10 seconds. The generated Haskell source lives under the newly created MAlonzo directory; namely `./MAlonzo/Code/CompilingAgda.hs`.

B.8 Absurd Patterns

When there are no possible constructor patterns, we may match on the pattern `()` and provide no right hand side —since there is no way anyone could provide an argument to the function. For example, here we define the datatype family of numbers smaller than a given natural number: `fzero` is smaller than `suc n` for any `n`, and if `i` is smaller than `n` then `fsuc i` is smaller than `suc n`.

Finite Types

```
{- Fin n  ≅  numbers i with i < n -}
data Fin : ℕ → Set where
  fzero : {n : ℕ} → Fin (suc n)
  fsuc  : {n : ℕ}
        → Fin n → Fin (suc n)
```

For each n, the type `Fin n` contains n elements; e.g., `Fin 2` has elements `fsuc fzero` and `fzero`, whereas `Fin 0` has no elements at all.

Using this type, we can write a safe indexing function that never "goes out of bounds".

Safe Indexing

```
_‼_ : {A : Set} {n : ℕ} → Vec A n → Fin n → A
[] ‼ ()
(x :: xs) ‼ fzero  = x
(x :: xs) ‼ fsuc i = xs ‼ i
```

When we are given the empty list, `[]`, then `n` is necessarily `0`, but there is no way to make an element of type `Fin 0` and so we have the absurd pattern. That is, since the empty type `Fin 0` has no elements there is nothing to define —we have a definition by *no cases*.

Logically "anything follows from false" becomes the following program[1]:

Ex Falso Quod Libet

```
data False : Set where

magic : {Anything-you-want : Set} → False → Anything-you-want
magic ()
```

Starting with `magic x = ?` then casing on `x` yields the program above since there is no way to make an element of `False` —we needn't bother with a result(ing right side), since there's no way to make an element of an empty type.

[1]Latin for: *From falsehood —ex falso— anything (lit: whatever you wish) follows —quodlibet.* Also known as "the principle of explosion".