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

January 28, 2021

PhD Thesis

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

Abstract

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

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

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

Put simply, the thesis is about making tedious *and* inexpressible patterns of programming in DTLs (dependently typed languages) become mechanical *and* expressible. The core of this thesis consists of Chapters 3 and 5, the first covering the first half of the items below and the second covering the second half. The formal foundation is covered in Chapter 2.

0. There are specific things we do as people to get things done.

1. These specific things are tedious and mostly mechanical.

2. Agda cannot let us accomplish these goals directly.

3. However, Agda allows a **pragmatic** approach to do so by rolling-out our own approach to modules: `Context`.

4. It's pragmatic since it's an Agda library *using* monadic `do`-notation and using Agda's type checking as well as all of its features, such as termination & well-definedness checking. The cost of such a library is that it is not as smooth as a built-in grouping mechanism.

5. Finally, the *idea* is sufficiently generic to be implemented in other DTLs; so the solution to (2) generalises to any DTL sufficient for implementing Context —the stronger the DTL, the more of `Context`'s combinators can be implemented, the more module combinators one has in hand. This idea is fleshed out in chapter 5 by providing the key insightful syntactic transformations on a DTL's grammar.

There are situations that occur often when working in a dependently-typed language, and it is reasonable enough to have the computer handle them (Chapter 4). In fact, we can have the computer do so without just string manipulation but instead using the DTL itself (Chapter 5).

A middle-path with margins

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

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

I would like you to be able to read this work *smoothly, with minimal interruptions.* As such, inspired by [41] among others, we have opted to include "mathematical graffiti" in the margins. In particular, the margins side notes may have *informal and optioniated* remarks[1]. We're trying to avoid being too dry, and aim at being somewhat light-hearted.

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

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

Dijkstra [27] might construe the graffiti as *mathematical politeness* that could potentially save the reader a minute. Even though a characteristic of academic writing is its terseness[2], we don't want to baffle or puzzle our readers, and so we use the informality of the graffiti to say what we mean bluntly, *but* it may be less accurate or not as formally justifiable as the text proper.

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

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

> *Some consider the puzzles that are created by their omissions as spicy challenges, without which their texts would be boring; others shun clarity lest their worth is considered trivial. [. . .] Some authors believe that, in order to keep the reader awake, one has to tickle him with surprises. [. . .] essential for earning the respect of their readership.*
> *—Edsger Dijkstra [27]*

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

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

A superficial cost of utilising margin space is that the overall page count may be 'over-exaggerated'[3]. Nonetheless, I have found long empty columns of margin space *yearning* to be filled with explanatory remarks, references, or somewhat helpful diagrams. Paraphrasing Hofstadter [47], the little pearls in the margins were so connected in my own mind with the ideas that I was writing about that for me to deprive my readers of the connection that I myself felt so strongly would be nothing less than perverse.

[3] Which doesn't matter, since you're likely reading this online!

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

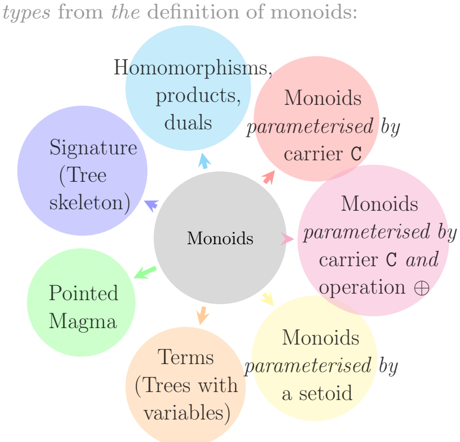
Contents

1. Introduction

The construction of programming libraries is managed by decomposing ideas into self-contained units we call 'packages'[0] whose relationships are then formalised as transformations that reorganise representations[1] of data. Depending on the *expressivity* of a language, packages may serve to avoid having different ideas share the same name —which is usually their *only* use— but they may additionally serve as silos of source definitions from which interfaces and types may be *extracted*. The figure to the right exemplifies the idea for monoids —which themselves model a notion of composition. In general, such derived constructions are *out of reach* from *within* a language and have to be extracted *by hand* by users who have the time and training to do so. Unfortunately, this is the standard approach; even though it is error-prone and disguises mechanical *library methods* (that are written *once* and proven correct) as *design patterns* (which need to be carefully implemented for *each* use and argued to be correct). The goal of this thesis is to show that sufficiently expressive languages make packages an interesting *and* central programming concept by extending their common use as silos of data with the ability for *users* to *mechanically* derive related ideas (programming constructs) as well as the relationships [34, 17] between them.

[0] Also known as 'modules'.

[1] Deriving related *types* from *the* definition of monoids:



[34] William M. Farmer. *A New Style of Proof for Mathematics Organized as a Network of Axiomatic Theories.* 2018. arXiv: 1806.00810v2 [cs.LO]

[17] Jacques Carette, William M. Farmer, and Michael Kohlhase. *Realms: A Structure for Consolidating Knowledge about Mathematical Theories.* 2014. arXiv: 1405.5956v1 [cs.MS]

When developing libraries, such as [53], in the dependently-typed language (DTL) Agda, one is forced to mitigate a number of hurdles. We turn to these hurdles in the following subsections —some of which are also discussed clearly in [18]. The remainder of this chapter is organised as follows: Sections 1.1 to 1.4 discussing the motivating problems[2] that arise when working in a DTL, then Section 1.5 briefly discusses our desire to have our resulting system be *usable*, and, finally, Section 1.6 concludes with an overview of the thesis as well as providing an estimate of the accessibility —interdependence— of the remaining chapters.

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

[18] 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

[2] Discussed in greater detail in Chapter 3.

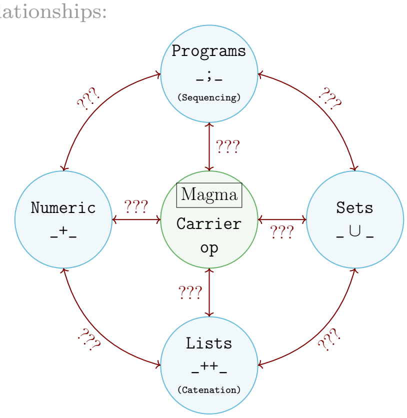
1.1. Practical Concern ♯1: Renaming and Remembering Relationships

There is excessive repetition in the simplest of tasks when working with packages; e.g., to *uniformly* decorate the names in a package with subscripts $_0$, $_1$, $_2$ requires the package's contents be listed thrice. It would be more economical[3] to *apply* a renaming[4] *function* to a package. Even worse, as shown to the right, sometimes we want to perform a renaming to view an idea in a more natural, concrete, setting; but shallow renaming mechanisms *lose the relationships* to the original parent package and so 'do nothing' coercions have to be written by hand.

The need to 'remember relationships'[5] is shared by the other concerns discussed in this section.

[3] Akin to the *decorations* of Z-notation.

[4] Given green, derive cyan candidate constructions, require red relationships:



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

1.2. Practical Concern ♯2: Unbundling

In general, in a DTL, *packages behave like functions* in that they may have a subset of their contents designated as *parameters exposed at the type-level* which users can *instantiate*. The shift between the two forms is known as **the unbundling problem**[6] [38]. Unfortunately, library developers generally provide only a few *variations* on *a* package; such as having no parameters or having only *functional symbols* as parameters[7] . Whereas functions can *bundle-up* or *unbundle* their parameters using currying and uncurrying, only the latter is generally supported and, even then, not in an elegant fashion. Rather than provide *several variations* on a package, it would be more economical to provide one singular fully-bundled package and have an operator that allows users to *declaratively*, "on the fly", expose package constituents as parameters.

Let us try to clarify this subtlety.

At its core, the unbundling problem is well-known as '(un)currying': The restructuring of record consuming functions as 'parameterised families of functions'. Uncurrying can be phrased as follows. [8]

$$\frac{A : \mathsf{Type} \qquad B : \mathsf{Type} \qquad C : \mathsf{Type}}{A \times B \to C \quad \cong \quad A \to (B \to C)}$$

[6] Here's a ubiquitous instance of this problem: Theorem statements
`∀ (G : Group) (M : Monoid)`
`→ Group.Carrier G ≡ Monoid.Carrier M`
`→ ···`
may be expressed more clearly —without an after-the-fact constraint— as
`∀ (C : Set)`
`(G : GroupOn C)`
`(M : MonoidOn C)`
`→ ··· .`
The problem is how to go from `Group` to `GroupOn`, and it numerous derivatives, *dynamically*.

[38] 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

[7] Recall the carrier `C` and operation `_⊕_` on page 7 on monoid constructions.

The right side brings a number of *practical conveniences* in the form of simplified concrete syntax —e.g., reduced parentheses for function arguments— and in terms of auxiliary combinators to 'fix' an A-value ahead of time —i.e., 'partial function application'. The *unbundling problem*[9] replaces simple product and function types with their *dependent* generalisations.[10]

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

As with currying, the right side here is preferable at times since it immediately[11] lets one 'fix' —i.e., select— a value $i_0 : I$ to obtain the specialised type

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

In contrast to the right, the left side can only be contorted[12] to simulate the idea of fixing a field, $i_1 : I$, ahead of time; e.g.:

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

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

The dependent nature of DTLs means that this problem is not solely about functions —and so, we cannot simply insist on formulations similar to the right side; i.e., omitting the record former 'Σ'. Since types can *depend on the values* of other types, this now becomes a problem about types as well. In particular, we may view the parameterised type family Z as being a new concept that is formed around a chosen substructure $i_0 : X$ —which must be referenced from 'outside' using the ambient structure Y; as shown in the informal 3-node diagram to the right. It would be far more practical to treat the structure we actually care about as if it were a 'top level item' rather than 'something to be hunted down'; as shown in the 2-node diagram to the right.

It is interesting to note that the unbundling problem appears in a number of guises within the setting of programming language design. For instance, it can be seen in numerous popular languages, including Haskell and JavaScript, in the form[13] of *pattern matching*, or *de-structuring*; wherein **explicit** treatment of record arguments as *packaging mechanisms,* **silently** disappears in the *presentation* of function definitions. Then, *implicit currying* is the feature that allows the presentation to accommodate arguments *sequentially* ("one at a time") rather than "all at once".

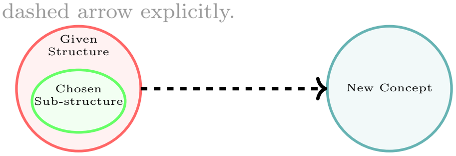
[8] The symbol '≅' means "isomorphic with" and it means "essentially interchangeable". More formally, it signals that there is a non-lossy protocol between two types. It is most generally defined in the setting of category theory: $A \cong B$ *precisely* when there are two transformations $f : A \to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

[9] Variations of this problem appear in various forms in computing; e.g., as *quantifier (un)nesting* in predicate logic or *lambda lifting* in programming language theory.

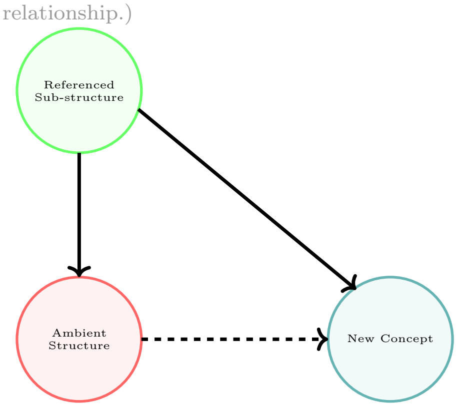
[10] Notice that *before* A, B, C were *independent* types; whereas *here* we have that Y depends on I and X, and X depends on I.

When we write $X : I \to \mathsf{Type}$ we are declaring that X is a *family of types indexed by the type I*. Dependent types and type-formers such as record-formation 'Σ' and parameterisation 'Π' are motivated in chapter 2.

[11] Unbundled forms: Obtain the dashed arrow explicitly.



[12] Bundled forms: Two solid arrows to get one dashed arrow: (In these diagrams, the arrows are used to denote a dependency relationship.)



1.3. Theoretical Concern ♯1: Exceptionality

DTLs blur the distinction between expressions and types, treating them as the same thing: *Terms*. This collapses a number of seemingly different language constructs into the same thing[14] Unfortunately[15] , packages are treated as *exceptional* values that differ from *usual* values —such as functions and numbers— in that the former are 'second-class citizens' which only serve to collect the latter 'first-class citizens'. This forces users to learn two families of 'sub-languages' — one for each citizen class. There is essentially no *theoretical* reason why packages do not deserve first-class citizenship, and so receive the same treatment as other *unexceptional*[16] values. Another advantage of giving packages equal treatment is that we are inexorably led to wonder what **computable algebraic structure** they have and how they relate to other constructs in a language; e.g., packages are essentially record-valued functions.

Perhaps the most famous instance of the promotion[17] of a second-class concept to first-class status comes from linear algebra, and subsequently, the theory of vector spaces. When there are a number of relationships involving a number of unknowns, the relationships could be 'massaged algebraically' to produce simper constraints on the unknowns, possibly providing 'solutions' to the system of relationships directly. The shift from *systems of equations* that serve to collect relationships, to *matrices* (expressing equations[18]) gave way to the treatment of such systems as algebraic entities unto themselves: They can be treated with nearly the same interface as that of integers, say, that of rings.[19] As such, 'component-wise addition of equations in system A with system B' becomes more tractable as $A + B$ and satisfies the many familiar properties of numeric addition. Even more generally, for any theory of 'individuals' one can consider the associated matrix theory —e.g., if M is a monoid, then the matrices whose elements are drawn from M *inherit* the monoidal structure— and so give a construction of *system of equations* on that theory. To investigate the algebraic nature of packaging mechanisms is another aim of this thesis.

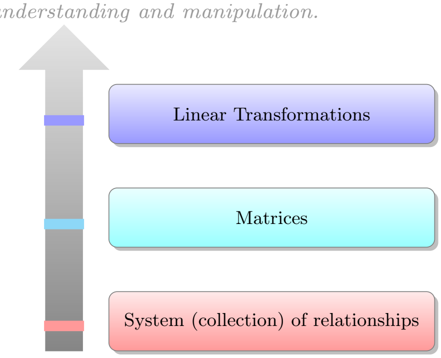
[13] Define `f : X × Y → Z` by projecting fields as needed `f p = ⋯ fst p ⋯ snd p ⋯` or by exposing the fields directly `f (x, y) = ⋯ x ⋯ y ⋯`. But to 'curry' is another matter: `f' = λ x • λ y • ⋯ x ⋯ y ⋯`.

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

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

[16] Differing from the usual, familiar.

[17] *With abstractions comes ease of understanding and manipulation.*



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

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

1.4. Theoretical Concern ♯2: Syntax

Packages, as we call them, serve to group together sequences of declarations. If any declarations are opaque, not fully defined, they become, what we call, *parameters* of the package —which may then be identified as a *record type* with the opaque declarations called *fields*. However, when a declaration is *intentionally opaque* not because it is missing an implementation, but rather it acts as a value construction itself then one uses *algebraic data types*, or 'termtypes'. Such types share the general structure of a package, as shown in the code block below, and so it would be interesting to illuminate the exact difference between the concepts —*if any*. In practice, one forms a record type to model an interface, instances of which are actual implementations, and forms an *associated* termtype to *describe computations* over that record type, thereby making available a syntactic treatment of the interface —textual substitution, simplification / optimisation, evaluators, canonical forms.

Spot the difference

Theory of monoids

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

Terms over 'variables' C

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

Binary trees with leaf labels drawn from C

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

Key insight:

$$\begin{array}{lcl} \_{⨾}\_ & \approx & \texttt{Branch} \\ \texttt{Id} & \approx & \texttt{Nil} \end{array}$$

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

"Termtype?"

We will refer to algebraic data types as *termtypes*, rather than *term type* or *term-type*. The reason for doing so is that in Chapter 2 we will discuss *terms* and *types*, and come to see them as indistinguishable —for the most part. As such, the phrase *term type* could be read ambiguously as "the type of terms" or as "the term denoting a type". For these reasons, we have chosen "termtype". Moreover, in Chapter 5, we will form a macro that consumes a particular kind of package and yields a termtype: The name of the macro is `termtype`.

1.5. Guiding Principle: Practical Usability

In this thesis, we aim to mitigate the above concerns with a focus on **practicality**. [20] A theoretical framework may address the concerns, but it would be incapable of accommodating *real-world use-cases* when it cannot be applied to real-world code. For instance, one may speak of 'amalgamating packages', which can always "be made disjoint", but in practice the union of two packages would likely result in name clashes —which could be avoided in a number of ways; i.e., selected, automatic, protocols— but the *user-defined names* are important and so a result that is "unique up to isomorphism" is not practical. As such, we will implement a framework to show that the above concerns can be addressed in a way that **actually works.** [21]

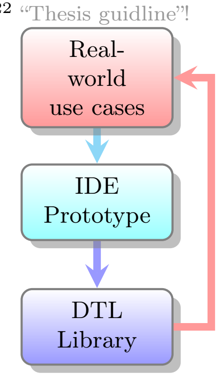
[20] If you can't use it, it's essentially useless!

[21] A concrete example is demonstrated later on, on page ??.

1.6. Thesis Overview

The remainder of the thesis is organised as follows. [22]

[22] "Thesis guidline"!



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

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

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

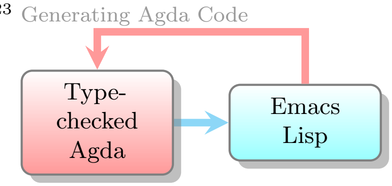
Chapter 3 consists of real world examples of problems encountered with the existing package system of Agda.

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

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

Unfortunately, the prototype[23] introduces a new sublanguage for users to learn. Packages are *nearly* first-class citizens: Their manipulation must be specified in Lisp rather than in the host language, Agda. However, the ability to rapidly, textually, manipulate a package makes the prototype an extremely useful tool to test ideas and implementations of package combinators. In particular, the aforementioned example of forming unions of packages is implemented in such a way that the amount of input required —such as *along* what interface should a given pair of packages be *glued* and *how* name clashes should be handled— can be 'inferred' (when not provided) by making use of Lisp's support for keyword arguments. Moreover, the union operation is a *user-defined* combinator: It is a *possible* implementation by a user of the prototype, built upon the prototype's "package meta-primitives".

[23] Generating Agda Code



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

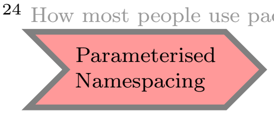
The prototype is given semantics as Agda types and functions by forming a **practical** library within Agda that achieves the core features of the prototype. The switch to a DTL is nontrivial due to the type system; e.g., fresh names cannot be arbitrarily introduced nor can syntactic shuffling happen without a bit of overhead. The resulting library is both usable and practical, but lacks the immense power of the prototype due to the limitations of the existing implementation of Agda's metaprogramming facility.

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

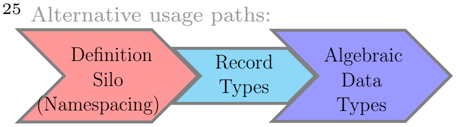
Chapter 6 concludes with a discussion about the results presented in the thesis.

The underlying motivation for the research is the conviction that packages play[24] *the* crucial[25] role for forming compound computations, subsuming *both* record types and termtypes.

[24] How most people use packages:



[25] Alternative usage paths:

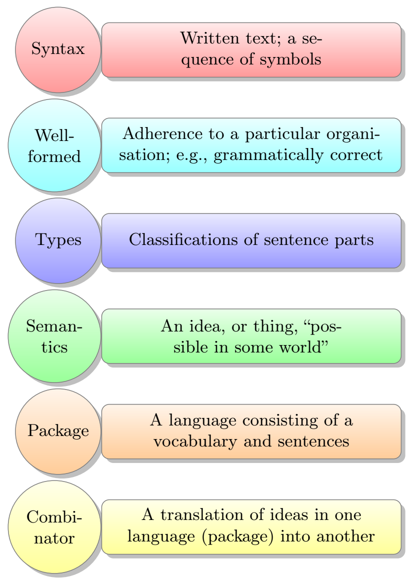


How accessible is this thesis?

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

2. Packages and Their Parts

The purpose of language is to communicate ideas that 'live' in our minds —conversely[0], language also influences the kinds of thoughts we may have. In particular, written text captures ideas independently of the person who initially thought of them. To understand the idea *behind* a written sentence, people agree on *how* sentences may be organised and *what* content they denote from their parts. For example, in English, a sentence is considered 'well-formed' if it is in the order subject-verb-object —such as *"Jim ate the apple"*— and it is considered 'meaningful' if the subject and object are noun phrases that *denote things in a world that could exist* and the verb is a *possible action* by the subject on the object. For instance, in the previous example, there *could* be a person named *Jim* who *could* eat an apple, and so the sentence is meaningful. In contrast[1] the phrase *"the colourless green apple kissed Jim"* is well-formed *but not* meaningful: The indicated action *could happen*, say, *in a world* of sentient apples; however, the subject —*the colourless green apple*— *cannot possibly exist* since a thing cannot be both lacking colour but also having colour at the same time.



[0]**Linguistics.** The idea that *language limits the kinds of thoughts one can have* is known as the Sapir-Whorf Hypothesis [82, 24, 67, 72] and it has largely been discredited in-preference to the weaker idea that language *influences* the kinds of thoughts one can have. For instance, in Arabic the singular word *akaltuha* tersely captures an idea, a sentence, that would require three words in English —namely, *I ate it.* For a computing example, in Prolog one may write a constraint solver —say to find a solution to a Suodku puzzle— which would require tenfold the number of lines in, say, Python since the former is intended to work with constraint problems. As such, thoughts can be had in different language, but some languages may allow thoughts to be more easily expressed.

[1]**Green Apples.** In our cursory glance of linguistic examples we spoke of *green apples* with the implicit understanding that *green* is an adjective that qualifies its subject, rather than *green apples* being taken as an atomic name of a *species* of apples that may not necessarily be green. That is, when we speak of $P\ x$ we mean an individual entity x that has the property P. This somewhat natural convention is superficially problematic in mathematics; so much so that it is dubbed *the red-herring principle*. Indeed, in mathematical practice, adjectives are often used to qualify their subjects in what seems like a contradictory fashion. For example, *a semigroup is a non-unitial monoid* is a terse summary of the, possibly unfamiliar, notion of semigroup using, the possibly more familiar, notion of monoid. However, a monoid, by definition, has a unit and so the phrase *non-until monoid* is technically meaningless; instead, it denotes the notion of a monoid with all references to a unit dropped, ignored. Interestingly, this use of adjectives to "dropping details" is a common combinator for producing new packages from old, as we will come to see.

Moreover, *depending on who you ask*, the action of the previous example —*the [. . .] apple kissed Jim*—, may be ludicrous *on the basis* that kissing is 'classified' as a verb whose subject, in the 'real' world, has the ability to kiss. As such, 'meaningfulness' is not necessarily fixed, but may vary. Likewise, as there is no one universal language spoken by all people, written text is also not fixed but varies; e.g., a translation tool may convert an idea *captured in* Arabic to a related idea *captured in* French. It is with these observations that we will discuss the concepts required to have a formal theory of packages, as summarised in the figure above.

Game-Play Analogy

The contents of the above figure are a bit abstract; so we reach for a *concrete* game-play based analogy that may make the concepts more accessible.

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

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

Machine

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

Colours

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

(The bold emphasis, on certain key words, below is *intended* as an informal **definition** of ideas to be fleshed out later in the chapter.)

Each of these is a **package**[2]: A sequence of 'declarations' of operations; wherein elements may be 'parameters' in the declarations of others. A **declaration** is a "name : classification" pair of words, *optionally* with another "name = definition" pair of words that shows how the new word *name* can be obtained from the vocabulary already declared thus far. For example, in these packages (languages) `thrice` and `purple` are aliases for expressions (sentences) constructed from other words. A **parameter** —also known as a **field**— is a declaration that is not an alias; i.e., it has no associated =-pair. Parameters are essentially the building blocks of a language; they cannot be expressed in terms of other words. A non-parameter is essentially *fully defined, implemented,* as an alias of a mixture of earlier words; whereas parameters are 'opaque' —*not yet implemented.* In particular, in the colours example above, `dark` *defines* a function that uses the *symbolic name* `mix` in its definition. There is an important subtlety between `mix` and `dark`: The latter, `dark`, is an *actual function* that is fully determined when an *implementation* of the *symbolic name* `mix` is provided. The (parameter) name `mix` is said to be a *function symbol* rather than a function: It is the *name* of a function, but it lacks any implementation and is thus not actually a function. A *function symbol* is to a function, like a name is to a person: Your name does not fully determine who you are as a person. # *reduce* the number of distinctions so that we have a *uniform* approach to

[2]**Interfaces.** By the end of the thesis, we hope the reader will see that there is *essentially no theoretical distinction* between: Packages, modules, classes, interfaces, records, and contexts. As such, we are *intentionally* using them as if they were synonymous —which contradicts popular usages. Briefly, a package with one parameter `p`, and declarations `ds` that may use the parameter, is essentially a function `λ p → ds'` that takes in a value for the parameter `p` and returns the package's declarations `ds` in the shape of a record of declarations `ds'`; finally, the main distinction between `p` and `ds` is that the declarations consist of a type declaration with associated definitions whereas `p` is only a type declaration lacking a definition, and so we treat packages as contexts " `p; ds` " and refer to non-definitional declarations as *parameters.* Traditionally, a 'parameter' refers to a part of the discussion that is allowed to vary and we are locally overriding the meaning of the word. Dear reader, whenever you read an article, the phrase "the author" changes it meaning, and so you have already encountered local overrides —even more so, when authors declare their conventions at the start of their papers; e.g., *for brevity, 'ring' means a commutative ring with one.*

This chapter is organised as follows. Section 2.1 sketches out the English sentences example from above —on colours— introducing the notation used for declaring grammars of languages, along with typing contexts. Section 2.2 then extrapolates the key insights using the idea of signatures. In section 2.3, the desire to present packages (signatures) *practically* in a uniform notation —to *reduce* the number of distinctions— leads to types that *vary* according to other types, thereby motivating Π-types; then the (un)bundling problem is used to motivate the introduction of Σ-type. Finally, the chapter concludes, in section 2.4, with a terse review the Agda language as a tool supporting the ideas of the previous subsections. In particular, the ideas of presented earlier in the chapter (Π, Σ, grammars) gain life in Agda as `record`s, namespacing `module`s, and algebraic `data`types (respectively).

Chapter Contents

2.1. What is a language?

In this section[3] we introduce two languages in preparation for the terminology and ideas of the next section. The first language, *Madlips*, will only be discussed briefly and is mentioned due to its inherit accessibility, thereby avoiding unnecessary domain specific clutter and making definitions clearer.

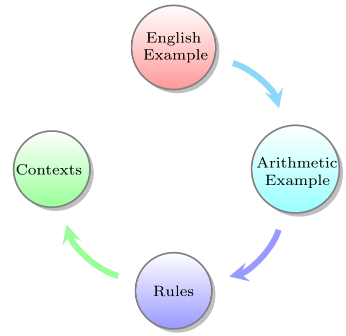
Madlips:[4] Simple English sentences have the form subject-verb-object such as *"Jim ate the apple"*. To *mindlessly* produce such sentences, one must produce a subject, then a verb, then an object —all from given lists of possibilities. A convenient notation to describe a language is its *grammar* [22, 23] presented in *Backus-Naur Form* [21, 43, 58, 57] as shown below.

The notation τ `::=` $\mathtt{c}_0$ `|` $\mathtt{c}_1$ `|` ... `|` $\mathtt{c}_n$ defines the name τ as an alias for the collection of words —also called *strings* or *constructors*— $\mathtt{c}_0$ or $\mathtt{c}_1$ or ... or $\mathtt{c}_n$; that is the bar '|' is read 'or'. The name τ is also known as a *syntactic category*. For example, in the Madlips grammar, `Subject` is the name of the collection of words *Jim*, *He*, and *Apple*. A constructor may be followed by words of another collection, which are called *the arguments of the constructor*. For example, the `Object` collection has a `The` constructor which must be followed by a word of the `Subject` collection; e.g, `The Apple` is a valid *value* of the `Object` collection, whereas `The` is just an incomplete construction of `Object` words. The last clause of `Object` is just `Subject`: An invisible (unwritten) constructor that takes a value of `Subject` as its argument; e.g., `He` and all other values of `Subject` are also values of the `Object` collection. Similarly, the `Sentence` collection consists of one invisible (unwritten) constructor that takes 3 arguments —a subject, a verb, and an object. Below is an example *derivation* of a *sentence* in the *language generated by this grammar*; at each '→' step, one of the collection names is replaced by one of its constructors until there are no more possible replacements —justifications are shown to the right.

Madlips Grammar

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

[3] The plan for this section is loosely summarised by the following diagram.



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

[22] Noam Chomsky. "A Note on Phrase Structure Grammars". In: *Inf. Control.* 2.4 (1959), pp. 393–395. DOI: 10.1016/S0019-9958(59)80017-6. URL: https://doi.org/10.1016/S0019-9958(59)80017-6

[23] Noam Chomsky. "On Certain Formal Properties of Grammars". In: *Inf. Control.* 2.2 (1959), pp. 137–167. DOI: 10.1016/S0019-9958(59)90362-6. URL: https://doi.org/10.1016/S0019-9958(59)90362-6

[21] R. I. Chaplin, R. E. Crosbie, and J. L. Hay. "A Graphical Representation of the Backus-Naur Form". In: *Comput. J.* 16.1 (1973), pp. 28–29. DOI: 10.1093/comjnl/16.1.28. URL: https://doi.org/10.1093/comjnl/16.1.28

Example Derivation

```
   Sentence
→ Subject Verb Object         -- Definition of 'Sentence'
→ Jim     Verb Object         -- Choose a 'Subject' value
→ Jim     Ate  Object         -- Choose a 'Verb' value
→ Jim     Ate  The Subject    -- Construct an 'Object' value
→ Jim     Ate  The Apple      -- Choose a 'Subject' value
```

Similarly, one may form `He Kissed Jim` as well as the meaningless[5] sentence `Apple Kissed He`.

◇ The first is vague, the pronoun 'He' does not designate a known person but instead "stands in" for a *variable*, yet unknown, person. As such, the first sentence can be assigned a meaning once we have a *context* of which pronouns refer to which people.

◇ The second just doesn't make sense. Sometimes nonsensical sentences can be avoided by restructuring the grammar, say, by introducing auxiliary syntactic categories. A more general solution is to introduce *judgement rules* that characterise the subset of sentences that are sensible.

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

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

Freshmen Grammar

```
Term ::= Zero | Succ Term | Term + Term    -- Numeric portion
       | True | False | Term ≈ Term        -- Boolean portion
```

Unlike the previous grammar, instead of `+ Term Term` to declare a constructor '+' that takes two `Term` values, we write the operation `_+_` *infix*[6], in the middle, since that is a common convention for such an operation. Likewise, `Term ≈ Term` specifies a constructor `_≈_` that takes two term values.

[43] Guoyong, Peimin Deng, and Jiali Feng. "Specification based on Backus-Naur Formalism and Programming Language". In: *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 - December 1, 2002, Proceedings*. 2002, pp. 95–101

[58] Jeroen F. J. Laros et al. "A formalized description of the standard human variant nomenclature in Extended Backus-Naur Form". In: *BMC Bioinform.* 12.S-4 (2011), S5. DOI: 10.1186/1471-2105-12-S4-S5. URL: https://doi.org/10.1186/1471-2105-12-S4-S5

[57] Donald E. Knuth. "backus normal form vs. Backus Naur form". In: *Commun. ACM* 7.12 (1964), pp. 735–736. DOI: 10.1145/355588.365140. URL: https://doi.org/10.1145/355588.365140

[5] We are treating sequences of symbols *extensionally* as mere representations, denotations, of unique ideas. For instance, in a context where `He` refers to `Jim`, we may as well say `He Ate The Apple` is *the same as* `Jim Ate The Apple`. However, the previous two Madlips sentences are *intrinsically*, by their very syntactic nature, *distinct*. Some operations are only possible when we treat sentences in one mode or the other; e.g., sentence decomposition is syntactic.

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

Example terms include the numbers `Zero`, `Succ Zero`, and `Succ Succ Zero` —which denote 0, 1 (the successor of zero), and 2 (the successor of the successor of zero). The sensible Booleans terms `True ≈ False` and `True` are also possible —regardless of *how true* they may be. However, the nonsensical terms `True + False` and `Zero ≈ True` are also possible. As mentioned earlier, judgement rules can be used to characterise the sensible terms: The relationship "term t is an element of kind τ", written `t` $: \tau$ is defined by (1) introducing a new syntactic category (called "types") to 'tag' terms with the kind of elements they denote, and (2) declaring the conditions under which the relationship is true.

Types for Freshmen

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

Judgement Rules

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

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

A rule " $\frac{premises}{conclusion}$ " means "if the top parts are all true, then the bottom part is also true" —for instance, in elementary school, one may have seen " $+\frac{\substack{11\\1}}{12}$ " for arithmetic—; some rules have no premises and so their conclusions are unconditionally true. That these are *judgement rules* means that a particular instance of the relationship `t` $: \tau$ is true if and only if it is the conclusion of 'repeatedly stacking' these rules on each other. For example, below we have a *derivation tree* that allows us to conclude the sentence `Zero ≈ Succ Zero` is a Boolean term —regardless of *how true* the equality may be. Such trees are both read and written from the *bottom to the top*, where each horizontal line is an invocation of one of the judgement rules from above, until there are no more possible rules to apply.

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

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

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

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

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

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

Extending Freshmen with Variables

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

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

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

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

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

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

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

Grammar for Contexts

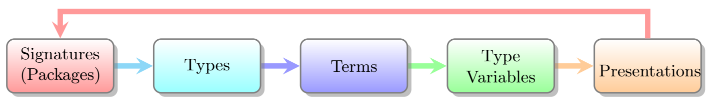
```
Context     ::= ∅ | Association, Context
Association ::= Variable : Type
```

Finally, it is interesting to observe that the addition of variables results in an interesting correspondence: *Terms in context are functions of their variables.* More precisely, if there is a method $[\![\_]\!]$ that *interprets* type names τ as actual sets $[\![\tau]\!]$ and terms `t` : τ as *values* of those sets $[\![\mathtt{t}]\!] : [\![\tau]\!]$, then a **term** in context $\mathtt{x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n \vdash \mathtt{t} : \tau$ corresponds to the **function** $f : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]; f(x_1, \ldots, x_n) = [\![t]\!]$. *That is, terms in context model parameterisation without speaking of sets and functions.* (Conversely, *functions* $A \to B$ "are" *elements* of B *in a context* A.) As mentioned in the introduction, we want to treat packages as the central structure for compound computations. To this aim, we have the approximate[7] slogan: ***Parameterised packages are terms in context.***

[7]Briefly, given a parameterised package in Agda (section 2.4) `module M (x : ℕ) where y : ℕ; y = 3 + x` we may form the term in context $x : \mathbb{N} \vdash y : \mathbb{N} = 3 + x$ (section 2.3.3) and there is a clear converse construction. Next, if we place a 'λ' in front of that context, we get a function, and so paramterised packages are functions.

2.2. Signatures

The languages of the previous section can be organised into *signatures*, which define interfaces in computing since they consist of the *names* of the types of data as well as the *names* of operations on the types —there are only symbolic names, not implementations. The purpose of this section is to organise the ideas presented in the previous section —shown again in the figure below— in a refinement-style so that the resulting formal definition permits the presentation of packages given in section 2.1 above.



The arrows “ $\mathcal{X} \longrightarrow \mathcal{Y}$ ” in the above diagram may be read as *“$\mathcal{X}$ give rise to an issue involving $\mathcal{Y}$”*. The purpose of this figure is to sketch out the intended transitions from signatures, to types, and, eventually, to presentations; then to an improved definition of *(generalised) signatures* which may be used as the formal definition of a *package*.

2.2.1. Typed terms in arbitrary signatures

A **signature**[8] [51, 3] is a tuple *($\mathcal{S}$, $\mathcal{F}$, src, tgt)* consisting of

- a set[9] $\mathcal{S}$ of *sorts* —the names of types—,
- a set[10] $\mathcal{F}$ of *(function) symbols*, and
- two mappings[11] $\mathtt{src} : \mathcal{F} \to \mathtt{List}\,\mathcal{S}$ and $\mathtt{tgt} : \mathcal{F} \to \mathcal{S}$ that associate a list[12] of *source sorts* and a *target sort* with a given function symbol.

Typing the symbols of a signature as follows[13] lets us treat signatures as general forms of ‘type theories’ since we may speak of ‘typed terms’.

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

Moreover, we regain the *typing judgements* of the previous section by introducing a grammar for *terms*. Given a set $\mathcal{V}$ of **variables**, we may define **terms**[14] with the following grammar.

[8] A signatures is also known as a *vocabulary*.
Unary Signatures are those with only one source sort for each function symbol —i.e., the length of $\mathtt{src}\, f$ is always 1— and so are just graphs. Hence, *signatures generalise graphical sketches*.
The slogan **Signatures** $\approx$ **Graphs** is captured by the following correspondence, (re)interpretation of signature components:

- Sorts $\approx$ “dots on a page”; Vertices
- Function symbols $\approx$ “lines between the dots”; Edges

[51] B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland, 1999

[3] S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, eds. *Handbook of Logic in Computer Science: Volume 5. Algebraic and Logical Structures*. Oxford University Press, Jan. 2001. DOI: 10.1093/oso/9780198537816.001.0001. URL: https://doi.org/10.1093%2Foso%2F9780198537816.001.0001

[9] In programming language communities [59], a signature is also known as *a programming language* —since our (non-generalised) signatures do not admit function types, they are not equivalent to $\lambda^{\times,\to}$, *a pure lambda calculus, with function and product types, and basic datatypes $\mathcal{S}$ and primitive operations $\mathcal{F}$*. Moreover, sorts are also known as *base types*, *type symbols*, or *type constants* —in constrast to ‘type constructors’, which are type-yielding functions.

[10] Sometimes signatures are presented with dedicated sets of ‘function symbols’ and ‘predicate, relation, symbols’. The chosen presentation avoids such a route since we want to use Agda, which does not distinguish between the two. Indeed, for us, *there are no predicate symbols nor function symbols, only symbols* and *there are no proof terms, only terms*. We may *simulate* predicate symbols by declaring that a sort, say, ‘$\mathbb{B}$’ in $\mathcal{S}$ models the Booleans, the truth-values —then, for instance, a ‘proof term’ is a term of type $\mathbb{B}$.

Grammar for Arbitrary Terms

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

Signature Typing

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

$$\frac{\Gamma \vdash t_1 : \tau_n \quad \ldots \quad \Gamma \vdash t_n : \tau_n \qquad f : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash \mathtt{f}\, t_1\, t_2 \ldots t_n : \tau}$$

As discussed in the previous section, variables are *not* necessary and if they are *not* permitted, we omit the first clause of `Term` and only use the second typing rule —we also drop the contexts since there would be no variables for which variable-type associations must be remembered. Without variables, the resulting terms are called *ground terms*. Since terms are defined recursively, inductively, the set of ground terms is non-empty precisely when at least one function symbol `c` needs no arguments, in which case we say `c` is a *constant symbol* and[15] make the following abbreviation:

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

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

2.2.2. Signature Presentation, Briefly

How do we actually **present** a signature?[16]

For instance, recall the Freshmen language, we can present an *approximation*[17] of it as signature by providing the necessary components $\mathcal{S}$, $\mathcal{F}$, `src`, and `tgt` as follows —where, for brevity, we write $\mathcal{B}$ and $\mathcal{N}$ instead of `Boolean` and `Number`.

```
𝒮 = {Number, Boolean}
ℱ = {Zero, Succ, Plus, True, False, Equal}
```

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

This is however rather *clumsy* and not that clear: We may collapse the `src, tgt` definitions into the `_:_→_` relation defined above; i.e.,

[11] Computing generally requires more sorts in comparison to traditional algebras in classical mathematics and so giving a signature's symbols with numbers for their arities is not as helpful as giving each symbol a list of its source sorts —whose length then determines its arity. For a traditional mathematical flavour, consult *A Course in Universal Algebra* [16].

[12] We write `List X` for the type of lists with values from `X`. The empty list is written `[]` and `[x`$_1$`, x`$_2$`, ..., x`$_n$`]` denotes the list of n elements `x`$_i$ from `X`; one says n is the *length* of the list.

[13] The wedge symbol '$\wedge$' is read "and"; e.g., $p \wedge q$ is read *"statements p and q are both true"*. The symbol '$\equiv$' is read "equivales", "exactly when", or "if and only if"; e.g., $p \equiv q$ is read *"p holds exactly when q holds"*.

[14] These are also known as *expressions* and *(abstract syntax) trees* —[59]: The leaves of which are labelled with variables (from $\mathcal{V}$) or constants (symbols f with `src` $f = []$), and the internal nodes are labelled with function symbols (from $\mathcal{F}$) of positive arity, with outdegree equal to the arity of the label. Hence, **abstract syntax is characterised algebraically using signatures**; moreover, every context-free grammar gives a canonical signatures —with non-terminals as sorts and constructors as function symbols— but the converse is not true since signatures may have infinitely many sorts or symbols, and they have no designated 'start state'.

[15] The second typing rule now becomes an axiom rather than inference rule: For any *constant* c of type τ:

$$\frac{}{\vdash\ c : \tau}\text{[Constant Type]}$$

The typing context is empty since the type of a constant is fixed, and therefore independent of the context in which it appears.

replacing *two* definition declarations `src Zero = []` $\wedge$ `tgt Zero = Number` by *one* definition declaration[18] `Zero : Number`. However, such a change would still leave function symbol names repeated twice: Once in the definition of $\mathcal{F}$ and once in the definition of `_:_→_`; the latter mentions all the names of $\mathcal{F}$ and so $\mathcal{F}$ may be *inferred* from the typing relationships. We are now left with two kinds of declarations: The sorts $\mathcal{S}$ and the typing declarations. However, the set $\mathcal{S}$ only serves to declare its elements as sort symbols; if we use a new relationship, say `_: Type` defined by τ `: Type` $\equiv \tau \in \mathcal{S}$, then the sort symbols can also be introduced by seemingly similar 'typing declarations'. With this approach, Freshmen can be introduced more naturally[19] as follows.

Freshmen as a Generalised Signature

```
Number  : Type
Boolean : Type

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

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

Notice, we started with two sets and two functions, i.e., signatures, but the above is a sequence of name-type associations. Recall, that the symbol Γ has consistently been used to denote such things. That is, these *'generalised' signatures are contexts.* We may thus define **packages** to be contexts where later declared names may be typed by earlier names; i.e., the types of later items may refer to the names of earlier declared items.

2.2.3. A grammar for types

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

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

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

[16] How do we *write down* the required parts of a signature? It is reasonable —'brute force'— to begin by presenting the required components of a signature as *listings*: The values of sets are listed out, and the value of function f at input x —$f(x)$— is shown in a table at the intersection of the row labelled f and the column labelled x. Are there better approaches?

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

[18] After all, the previous section sets up typed terms in any signature. That is, **replace `src`, `tgt` in preference to `_:_→_`.**

[19] It is important to note that there are three relations here with ':' in their name —`_:Type`, `_:_→_`, and `_:_` for constant-typing. These are summarised explicitly at the start of the next section.

Induced Grammar for Types

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

The type $\mathbb{1}$ is used for constants: With this grammar a constant $c : \tau$ would have type $\texttt{c} : \mathbb{1} \to \tau$. The symbol $\mathbb{1}$ is used simply to indicate that the function symbol `c` takes no arguments. The introduction of $\mathbb{1}$ saves us from having to account for the constant-typing relationship[20] as if it were a primitive predicate.

[20] Defined above by $\texttt{c} : \tau \equiv \texttt{src c = []} \wedge \texttt{tgt c} = \tau$.

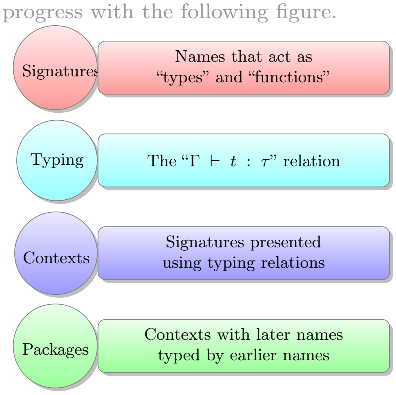
We may now form type *expressions, terms*, $\alpha \to \beta$ and $\alpha \times \beta$ but there is no way for the type β to depend on the type α. In particular, recall that in Freshmen we wanted to have $\texttt{s} \approx \texttt{t}$ to be a well-formed term of type `Boolean` *provided* `s` and `t` have the *same* type, either `Number` or `Boolean`. That is, $\_\approx\_$ wants to have *both* $\texttt{Number} \times \texttt{Number} \to \texttt{Boolean}$ *and* $\texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean}$ as types —since it is reasonable to compare either numbers *or* truth values for equality. But a function symbol can have only *one* type —since `src` and `tgt` are (deterministic) functions[21] . If we had access to variables which stand-in for types, we could type equality as $\alpha \times \alpha \to \texttt{Boolean}$ *for any type* α.

[21] A *function* is an association of ‘inputs’ to unique ‘outputs’.

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

Even though types *constrain* terms, there seems to be a subtle repetition: The `TYPE` grammar resembles the `Term` grammar. In fact, if we pretend `Type`, $\mathbb{1}$, $\_\times\_$, $\_\to\_$ *are* function symbols, then `TYPE` is subsumed by `Term`. Hence, we may conflate the two into one declaration to obtain *dependently-typed terms* —a concern which we will return to at a later time[22] .

[22] For now, we may summarise our progress with the following figure.



2.3. Presentations of Signatures —Π and Σ

Since a signature’s types also have a grammar, viz `TYPE`, we can present a signature in the natural style of “name : type-term” pairs. That is, a signature may be presented as a context; i.e., sequence of declarations δ_1, δ_2, …, δ_n *such that* each δ_i is of the form $\texttt{name}_i$: $\texttt{type}_i$ where $name_i$ are unique names but $type_i$ are *terms* from the `TYPE` grammar. *Conversely*[23] such a presentation gives rise to a unique signature $(\mathcal{S}, \mathcal{F}, \texttt{src}, \texttt{tgt})$ where:

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

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

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

This is one of the first instances of a syntax-to-semantics relationship: **A context is a syntactic representation of a (generalised) signature.** [24] However, with a bit of experimentation one quickly finds that the syntax is "too powerful": There are contexts that do *not* denote signatures. Consider the following grammar which models 'smart' people and their phone numbers. Observe that the 'smartness' of a person *varies* according to their location; for example, in, say, a school setting we have 'book smart' people whereas in the city we have 'street smart' people and, say, in front of a television we have 'no smart' people. Moreover, the function symbol `call` for obtaining the phone number of a 'smart person' must necessarily have a variable that accounts for how the smart type *depends* on location. However, if variables are not permitted, then `call` cannot have a type —which is unreasonable: We do not need *arbitrary* stand-ins, but rather *local* pronouns, variables. It is a well-defined context, but it does not denote a signature[25] .

Calling-smart-people Context

```
Location : Type

School   : Location
Street   : Location
TV       : Location

Smart    : Location → Type

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

The first problem, the type of `Smart`, is easily rectified: We take the sorts $\mathcal{S}$ to be *all* names τ in the context that produce a TYPE term; i.e., those names τ for which there exists a sub-context Γ such that $\Gamma \vdash \tau : \mathtt{Type}$. Sorts now may *vary* or *depend* on other sorts.

[23] **Proof of the claim:**

1. By induction on the number n.
2. When $n = 0$, there are no declarations and the outline construction yields the fully empty signature $(\emptyset, \emptyset, \emptyset, \emptyset)$.
3. When $n \geq 1$, let δ_n be the final declaration. Then by induction the previous n - 1 declarations constitute a signature $(\mathcal{S}', \mathcal{F}', \mathtt{src}', \mathtt{tgt}')$. Decompose $\delta_n = (\eta : \tau)$. There are two cases to consider.
   a) τ = `Type`: Since we assumed the names are unique,

   we have $\eta \notin \mathcal{S}'$ and so $(\mathcal{S}' \cup \{n\}, \mathcal{F}', \mathtt{src}', \mathtt{tgt}')$ is a signature.

   a) $\tau \neq$ `Type`: It must thus be a construction involving one of '→, ×, 𝟙'; by definition of the TYPE *assuming no variables*. In any case, we have a function symbol. Since we assumed the names are unique, we have $\eta \notin \mathcal{F}'$ and so $\mathtt{src}', \mathtt{tgt}'$ do not assign any type to η. Hence, we may define $\mathtt{src}\, s$ to be $\mathtt{src}' s$ unless $s = \eta$ in which case we yield the antecedent of τ if any, or 𝟙 otherwise. Likewise, define `tgt` to behave as `tgt′` except for η in-which case yield the consequent of τ if any, or all of τ otherwise.

[24] Signatures are all syntax; so we are interpreting contexts as a syntax for another syntax (signatures).

[25] *Ignoring* `Smart` and `call`, the figure to the left yields the following signature.

- ◇ $\mathcal{S}$ = `{Location, Phone}`
- ◇ $\mathcal{F}$ = `{School, Street, TV}`
- ◇ $\mathtt{src}\, f = []$, for all $f : \mathcal{F}$, and
- ◇ $\mathtt{tgt}\, f = \mathsf{Location}$, for all $f : \mathcal{F}$.

2.3.1. Motivating the need for Π and Σ

The second problem, the type of `call`, requires the introduction of a new[26] type operation. The operation $\Pi\_:\_\bullet\_$ will permit us to type function symbols that have variables in their types even when there is no variable collection $\mathcal{V}$.

[26] Those familiar with set theory may remark that dependent types are not *necessary* in the presence of power sets: Instead of a *single* name `call`, one uses a (possibly infinite) *family of names* call_ℓ for each possible name ℓ. Even though power sets are not present in our setting, dependent types provide a natural and elegant approach to *indexed types* in lieu of an encoding in terms of *families of sets or operations.* Moreover, an encoding *hides* essential features of an idea such as dual concepts: Σ and Π are 'adjoint functors'. Even more surprising, working with Σ and Π leads one to interpret "propositions as types" with predicate logic quantifiers ∀/∃ encoded via dependent types Π/Σ; whence the slogan:

"Programming ≈ Proving"

Dependent Function Type

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

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

The *values* of function types are expressed as λ `x` : τ • `t`; this *denotes* the function that takes input `x` : τ and yields output `t`. One then writes `f e`, or `f(e)`, to denote the application of the function `f` on input term `e`.

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

Before extending the previous definition of formal signatures, there is a practical[27] subtlety to consider. Suppose we want to talk about smart people *regardless* of their location, how would you express such a type? The type of `call` : (Π l : `Location` • `Smart` l → `Phone`) reads: *After picking a particular location ℓ, you may get the phone numbers of the smart people at that location.* More specifically, `Smart` ℓ is the type of smart people **at a particular** location ℓ. Since, in this case, we do not care about locations, we would like to simply pick a person who is located **somewhere**. The ability to "bundle away" a varying feature of a type, instead of fixing it at a particular value, is known as the **(un)bundling problem**[28]. It is addressed by introducing a new[29] type operator $\Sigma\_:\_\bullet\_$ —the symbol 'Σ' is conventionally used both for the name of signatures and for this new type operator.

[27] Motivating Σ!

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

[29] The Σ-types denote disjoint unions and are sometimes written as $\coprod$ —the 'dual' symbol to Π.

Difference between Π and Σ

`Π ℓ : Location • Smart ℓ`	"Pick a location, then pick a person"
`Σ ℓ : Location • Smart ℓ`	"Pick a person, who is located *somewhere*"

More generally,

`Π a : A • B a`	"Pick a value `a : A`, to get `B a` values"
`Σ a : A • B a`	"Pick a value `b : B a`, which is tagged by *some* `a : A`"

Dependent Product Type

$$\begin{array}{ll} & \Sigma\, a : A \;\bullet\; B\, a \\ \equiv & \text{“Pairs } (a, b)\text{, with } a : A \text{ and } b \text{ is a value of } \textit{type } B\, a\text{”} \end{array}$$

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

Thinking of type families $B : A \to$ `Type` as *predicates* or *constraints*, or *interfaces*, then one may think of $B\, a$ as the collection of *proofs* of the proposition $B\, a$, or as a *witness* to the constraint, or as an implementation to the interface. As such, Σ-types $\Sigma\, a : A \bullet B\, a$ are sometimes denoted using set notation $\{a : A \mid B\, a\}$ ('refinement types') and using logical notation $\exists a : A \bullet B\, a$.

The *values* of product types are expressed as `(x , w)`; this *denotes* pair of items where the second may depend on the first. One then writes `let (x, w)` $\coloneqq \beta$ `in e` to 'unpack' the pair value β as the pair `(x, w)` for use in term `e`.

Old ideas as abbreviations: The type operator `_→_` did not accommodate dependence but Π does; indeed if B does not depend on values of type A, then $\Pi a : A \bullet B$ is just `A → B`. Likewise[30] , Σ generalises `_×_`. That is, provided B is a type that does not vary:

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

[30] Since Π/Σ are the *varying* generalisations of →/×, sometimes Π/Σ are written as $(a : A) \to B\, a$ and $(a : A) \times B\, a$, respectively.

2.3.2. Examples: Π/Σ or $\to/\times$

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

Example 1: People and their birthdays

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

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

Example functions in this type are f and g below…

```
f Monday  = Jim
f Tuesday = Alice

g Monday  = Mark
g Tuesday = Alice
```

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

Person	Birthday
Jim	Monday
Alice	Tuesday
Mark	Monday

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

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

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

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

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

Example 2: English words and their lengths

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

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

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

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

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

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

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

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

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

Example 3: "All errors are type errors"

Suppose `get i xs` is the i-th element in a list `xs = [`x_0`, `x_1`, ..., `x_n`]`, what is the type of such a method `get`?

Using `get : Lists → ℕ → Value` will allow us to write `get [`x_1`, `x_2`] 44` which makes no sense: There is no 44-th element in that 2-element list! Hence, the `get` operation must constrain its numeric argument to be at most the length of its list argument. That is, `get : (Π (xs : Lists) • ℕ< (length xs) → Value)` where `ℕ< n` is the collection of numbers less than n. *Now the previous call, `get [`x_1`, `x_2`] 44` does not need to make sense since it is ill-typed*: The second argument does not match the required constraining type.

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

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

Proof Sketch

Suppose we want to avoid the erroenous situation **E** which can be expressed in higher order logic. Then we can type our program so that its output type is a dependent product $\Sigma\, o : \mathbf{O} \bullet \neg\mathbf{E}$, involving the intended output type **O** and a proof obligation —i.e., a value, witnessing the impossibility of **E**.

2.3.3. Defining Generalised Signatures

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

For any set of 'names' $\mathcal{U}$, suppose[31] $\mathsf{Term}_{\mathcal{U}}$ is a set of 'terms'[32]. Moreover, suppose: (1) Every name is a term; i.e., $\mathcal{U} \subseteq \mathsf{Term}_{\mathcal{U}}$. (2) There is a dedicated[33] name `Type`. (3) $\mathsf{Term}_{\mathcal{U}}$ is endowed with a "typing judgement relation $\_ \vdash \_ : \_$"; i.e., a ternary predicate on 'contexts'-'terms'-'types'— a 'context' is a list of name-to-term pairs and a 'type' τ is any term for which there is some context Γ and term `t` such that $\Gamma \vdash t : \tau$. We refer[34] to such triples $(\mathcal{U}, \mathsf{Term}_{\mathcal{U}}, \_ \vdash \_ : \_)$ as **generalised type theories**[35] (GTT).

GTTs allow us to speak of arbitrary typed expressions and varying degrees of actual typing. For instance, as previously discussed, every signature gives rise to a typing relation that ignores any presence of variables. However, GTTs are strictly more powerful than classical signatures since they allow not only nullary types (primitive sorts), but also *type constructors* and *dependent-types*: When Γ is a minimal context such that $\Gamma \vdash \tau$: `Type` then we say τ is a **(nullary) type** precisely when Γ is empty, and otherwise speak of a **type constructor, construction**; moreover, if Γ associates variables to terms besides `Type`, then we speak of a **dependently-typed construction**.

For instance, let $\mathcal{U} = \{A\}$ and let Term be the set generated by the following grammar[36].

Term grammar for an example GTT

```
Term  ::=  𝒰  |  ℕ  |  Vec Term Term
```

Finally, we may take the typing relation to be generated by two clauses, for any context Γ: (1) $\Gamma \vdash \mathbb{N}$: `Type` and (2) Γ, τ : `Type`, `n` : $\mathbb{N} \vdash$ `Vec` τ `n` : `Type`. If we take Γ to be the empty context, we find that $\mathbb{N}$ is a (nullary) type, whereas `Vec` is a type construction —in fact, a dependent type, since the minimal context required to type it associates the variable `n` to the non-`Type` term $\mathbb{N}$. Moreover, the typing relation does not associate a type with any names (variables) of $\mathcal{U}$, but[37] *under the supposition* that the variable name `A` were typed `Type`, and `n` is typed $\mathbb{N}$, then `Vec A n` would be a type.

Informally, in our exploratory investigation into a convenient *presentation* of signatures, we were inexorably led to having later declared types depend on earlier types. Likewise, the previous GTT example could be rendered as follows:

[31] The subscript is omitted when there is no ambiguity.

[32] Any collection, possibly generated by a grammar.

[33] It serves to provide a uniform way to identify 'types' —uniform in that it mirrors the way values are typed. Otherwise we would need a *dedicated* predicate, such as $\_\vdash\_$ `:Type` from the previous section. It answers the question "Some terms are types, how do we find them?"

[34] A **variable** is a name `x` of $\mathcal{U}$ for which $\Gamma \vdash$ `x` : τ can only happen when Γ contains the association of `x` to τ; i.e., a variable is a name about which information is known *only when the information is hypothesised*. A non-variable is known as a **value** or *well-defined name*. If $\Gamma \vdash$ `t` : τ and $\Gamma \vdash \tau$: `Type` we refer to `t` as an **expression** or **term**, to τ as a **type**, and to `Type` as a **kind**. More accurately, when Γ is a minimal context such that $\Gamma \vdash \tau$: `Type` then we say τ is a **type** precisely when Γ is empty, and otherwise speak of a **type constructor, construction**; moreover, if Γ associates variables to terms besides `Type`, then we speak of a **dependently-typed construction** —e.g., Π and Σ. This is important enough that it occurs in the main text and in the margin.

[35] An example is shown in the next section!

[36] As done before, the first clause of this grammar is an invisible constructor injecting names of $\mathcal{U}$ into the set of terms.

[37] Since this example's typing relation is inductively defined, such a supposition is absurd.

Example: An entire GTT viz a single context

```
ℕ   : Type
Vec : Type → ℕ → Type
```

We regain a canonical GTT from such a presentation as follows: (0) The name set $\mathcal{U}$ is the infinitely countable set of strings formed from all possible non-whitespace written ligatures, which includes the set of all names preceding the first ':' in each line of the presentation. The set $\mathsf{Term}\mathcal{U}$ is defined inductively by the next two clauses. (1) All names are included in the set of terms $\mathsf{Term}\mathcal{U}$. (2) Names for which the right side of the ':' contains n occurances of the '→' symbol are constructors that (inductively) consume n arguments of the term set being defined. (3) Finally, the typing relation _⊢_:_ is defined inductively with clauses

$$\Gamma\texttt{,}\ \texttt{t}_1 : \tau_1\texttt{,}\ \texttt{t}_2 : \tau_2\texttt{,}\ \ldots\texttt{,}\ \texttt{t}_n : \tau_n \vdash \eta\ \texttt{t}_1\ \texttt{t}_2\ \ldots\ \texttt{t}_n : \texttt{Type}$$

for every declaration[38] $\eta : \tau_1 \to \tau_2 \to \cdots \to \tau_n \to \texttt{Type}$.

That we are able to reconcile our presentation language with a sound formalising is promising. However, as it stands, our GTT example has `Vec` *built-in*, statically, and the only thing that can vary —with respect to that example— is the collection of variables[39] . It would be nice if we had a way to *append* GTTs with extra structure as we see fit; e.g., to dynamically declare names to be new types or type constructions or members of a type. Such 'dynamically extendable GTT-like structures' are what we have been calling *generalised signatures*.

A **generalised signature**, with respect to a chosen GTT $(\mathcal{U}, \mathsf{Term}_{\mathcal{U}}, \_ \vdash \_ : \_)$, is a set of triples[40] $(\beta_i, \Gamma_i, \tau_i, \delta_i)$ where the β_i are *unique* names drawn from $\mathcal{U}$, the Γ_i are name-to-term associations, the τ_i are terms, and the δ_i are either terms or the special symbol '-'. One then *extends* the underlying typing judgement by the rules $\overline{\Gamma_i \vdash \beta_i : \tau_i}$, *and then* ensures the resulting system is *coherent*:

1. The claimed types are recognised by the theory as types: $\Gamma_i \vdash \tau_i : \texttt{Type}$ for all i;

2. Definitions match types: $\Gamma_i \vdash \delta_i : \tau_i$ for all i;

3. Types are unique; i.e., whenever $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \tau'$ then[41] $\tau \equiv \tau'$ —we will return to propositional equality in a later section.

Due to the latter two coherence conditions, the tuples $(\beta_i, \Gamma_i, \tau_i, \delta_i)$ are *presented*[43] as $\beta_i : \Gamma_i \to \tau_i = \delta_i$ when δ_i is not the special symbol '-' and otherwise presented as $\beta_i : \Gamma_i \to \tau_i$.

[38] When $n = 0$, we have declarations η : `Type` and so typing judgements $\Gamma \vdash \eta$: `Type`.

[39] It can't vary much if we use all ligatures!

[40] Alternatively, we have a triple $(\mathcal{B}, \mathsf{type}, \mathsf{definition})$ where $\mathcal{B} \subseteq \mathcal{U}$, $\mathsf{type} : \mathcal{B} \to \mathsf{Context} \times \mathsf{Term}_{\mathcal{U}}$, and $\mathsf{definition} : \mathcal{B} \to \mathsf{Term}_{\mathcal{U}}$ is a *partial* function. Then one sets $\mathcal{B} = \{\beta_i\}_i$ and $(\Gamma_i, \tau_i) = \mathsf{type}\,\beta_i$ and $\delta_i = \mathsf{definition}\,\beta_i$ if defined or '-' otherwise.

We *interpret* `Type` as the type of all types; whereas the β_i let us *suppose* a collection of *names* for either types/sorts *or* function symbols, and they may be *aliases* to existing terms δ_i.

[41] To allow subtyping, inclusion instead of equality would be required.

[43] We are now overloading the existing colon ':' relation to be part of a mixfix name, _ : _ → _ = _ to denote tuples. The use of contexts this way occurs later as **telescopes** when we get to Agda. Another reasonable notation would be $\Gamma_i \vdash \beta_i : \tau_i = \delta_i$, overloading the judgement relationship name.

For instance, continuing with the previous GTT example, we can form a generalised signature with the two *tuples* $\mathbb{B}$: `Type` ⊢ `pit` : $\mathbb{B}$ and ⊢ $\mathbb{B}$: `Type` . Notice that the formal tuples are not as economical as the sequential line-by-line presentation, due to the repetition of the newly minted value $\mathbb{B}$: `Type`. Moreover, note that $\mathbb{B}$ is a *value* in the second tuple —since, by definition, the name $\mathbb{B}$ is typeable—; however, if we omit the first clause, then $\mathbb{B}$ is, by definition, a variable and we have declared `pit` to be a polymorphic value of any given type.

In summary, *a generalised signature extends a generalised type theory by declaring some names to be values (such as type constructions) and possibly outright defining them explicitly.* Crucially, a generalised signature may be presented as a sequence of declarations $d_1, \ldots, d_n$ where each d_i is of the form "*name* : *term* = *term*" where the "= *term*" portion is optional and the names are unique. When presented with multiple lines, we replace commas by newlines, and split "*name* : *type* = *definition*" into two lines: The first being "*name* : *type*" and the second[42] , if any, being "*name* = *definition*".

2.3.4. MLTT: An example generalised type theory

A portion[44] Martin-Löf Type Theory (MLTT)[45] [3, 8] is presented as the GTT having the terms generated inductively by the grammar and rules below —for any set of names $\mathcal{U}$.[46]

Generalised Terms

```
Term
    ::= x            -- A "variable, name"; a value of 𝒰
     | Type          -- The type of types
     -- For previously constructed types τ and τ',
     -- previously constructed terms tᵢ,
     -- and variable name x:
    | (Π x : τ • τ') | (λ x : τ • t)          |  t₁ t₂
    | (Σ x : τ • τ') | let (t₁, t₂) ≔ t₃ in t₄ | (t₁, t₂)
```

The rules[47] below classify the well-formed generalised terms.

. . . .

First are rules about contexts in general. For instance, the second rule[48] says *if* Γ *associates* x *to* τ*, then indeed it does so.* The third rule[49] *introduces new names* into a context.

$$\frac{}{\Gamma \;\vdash\; \texttt{Type} : \texttt{Type}}\text{[Type-in-Type]}$$

[42] In the next example, MLTT, declarations of functions `name = (λ x : τ • e)` are instead simplified to `name x = e` .

[44] Only a portion is shown since we will cover the omitted features in the section 2.4, using Agda.

[45] On which Agda is based.

[3] S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, eds. *Handbook of Logic in Computer Science: Volume 5. Algebraic and Logical Structures.* Oxford University Press, Jan. 2001. DOI: 10.1093/oso/9780198537816.001.0001. URL: https://doi.org/10.1093%2Foso%2F9780198537816.001.0001

[8] Roland Carl Backhouse and Paul Chisholm. "Do-It-Yourself Type Theory". In: *Formal Aspects Comput.* 1.1 (1989), pp. 19–84. DOI: 10.1007/BF01887198. URL: https://doi.org/10.1007/BF01887198

[46]

- $\mathcal{U}$ and `Type` together form the "sort structure"
- Π, λ, and (the invisible) application form the "functional structure"
- Σ, `let`, and tupling form the "record/packaging structure"

Recall: If `t : τ` and `τ : Type` we refer to `t` as an **expression**, to τ as a **type**, and to `Type` as a **kind**.

[47] There are numerous other useful rules, which we have omitted for brevity.

[48] The Variables rule is also known as Assumption or Reflexivity and may be rendered as follows.

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

[49] The weakening rule is helpful for ignoring "unnecessary" assumptions.

$$\frac{\Gamma(x) = \tau}{\Gamma \ \vdash \ x : \tau}[\text{Variables}]$$

$$\frac{\Gamma \vdash t : \tau \qquad x \text{ is not a name in } \Gamma \qquad \Gamma \vdash \alpha : \mathtt{Type}}{\Gamma, x : \alpha \vdash t : \tau}[\text{Weakening}]$$

Next[50] are the rules for dependent functions.

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

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

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

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

Then[51] the rules for dependent sums.

[51] Just as Σ is the dual to Π, in some suitable sense, so too the *eliminator* `let` is dual to the *constructor* lambda λ.

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

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

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

Finally, provided B is a type that does not vary; i.e., the variable `x` does not occur in B,

$$\frac{\Gamma \ \vdash \ t : A \times B}{\Gamma \ \vdash \ t : (\Sigma\, x : A \bullet B)}[\text{Abbreviation}]$$

$$\frac{\Gamma \ \vdash \ t : A \to B}{\Gamma \ \vdash \ t : (\Pi\, x : A \bullet B)}[\text{Abbreviation}]$$

. . . .

The rules for Π and Σ show that they are *families* of types 'indexed' by the first type. The rules only allow the construction of types and variable values, to construct *values of types* we will need some starting base types, whence the need[52] for signatures.

Π and Σ together allow the meta-language to be expressed in the object-language

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

Π types *internalise* contexts

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

Σ types *internalise* pairing contexts

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

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

Pointed Families

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

This is a generalised signature *within* the above GTT.

Since the names are completely new and there are unique declarations for each name, we have unique types; moreover since there are no definitions, and so there is only one condition to check in order to satisfy the required coherency constraint on generalised signatures. Namely, there the claimed types are actually recognised as types by the underlying theory *after* we extend the typing judgement with these new relationships; i.e., we need to show:

1. $\vdash$ `Type` : `Type` —since Γ_1 is the empty context and $\tau_1 =$ `Type`.

2. $\vdash (A \to \mathtt{Type}) : \mathtt{Type}$ —since Γ_2 is the empty context.

3. $\vdash (\Pi a : A \bullet B\,a) : \mathtt{Type}$

The first is just the Type-in-Type rule, the second is a mixture of the Abbreviation and Π-Formation rules; the third one is the most involved, so we verify it as an example derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash B : A \to \mathtt{Type}}\,[\text{Declaration}]}{a : A \vdash B : A \to \mathtt{Type}}\,[\text{Weak}]}{a : A \vdash B : (\Pi a : A \bullet \mathtt{Type})}\,[\text{Abbrev}] \qquad \dfrac{}{a : A \vdash a : A}\,[\text{Vars}]}{a : A \vdash B\,a : \mathtt{Type}}\,[\Pi\text{-Elim}]}{\vdash (\Pi a : A \bullet B\,a) : \mathtt{Type}}\,[\Pi\text{-Intro}]$$

Signatures are a staple of computing science since they formalise interfaces and generalise graphs and type theories. Our generalised signatures have been formalised “after the fact” from the creation of the prototype for packages —see Chapter 4. In the literature, our definition of generalised signatures is essentially a streamlined presentation of Cartmell’s ‘generalised algebraic theories’[53,53] expect that we do not allow arbitrary equational ‘axioms’ instead using “name = term” rather than “term = term” axioms which serve as *default implementations* of names. Support for default definitions is to place the prototype on a sound footing, but otherwise we do not make much use of such a feature outside that chapter.

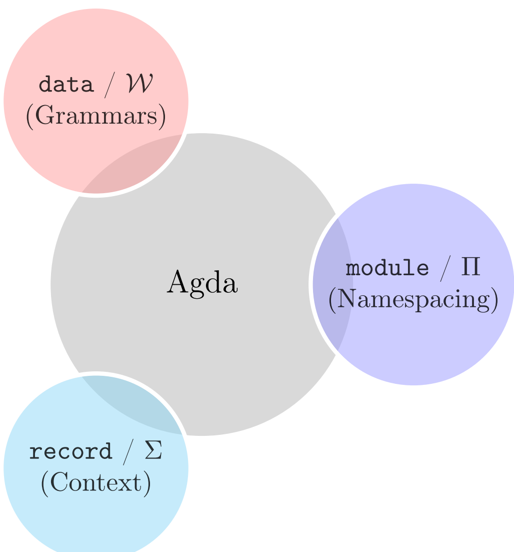
> Readers familiar with elementary computing may note that our contextual presentations, when omitting types, are essentially “JSON objects”; i.e., sequences of key-value pairs where the keys are operation names and the values are term descriptions, possibly the “null” description “–”.

[53] John Cartmell. “Generalised algebraic theories and contextual categories”. In: *Ann. Pure Appl. Log.* 32 (1986), pp. 209–243. DOI: `10.1016/0168-0072(86)90053-9`. URL: `https://doi.org/10.1016/0168-0072(86)90053-9`

[53] Quoting Cartmell: *Thus, a generalised algebraic theory consists of (i) a set of sorts, each with a specified role either as a constant type or else as a variable type varying in some way, (ii) a set of operator symbols, each one with its argument types and its value type specified (the value type may vary as the argument varies), (iii) a set of axioms. Each axiom must be an identity between similar well-formed expressions, either between terms of the same possibly varying type or else between type expressions.*

2.4. A Whirlwind Tour of Agda

We have introduced a number of concepts and it can be difficult to keep track of when relationships $\Gamma \vdash t : \tau$ are in-fact derivable. The Agda[54,55,56,57] programming language will provide us with the expressivity of generalised signatures and it will keep track of contexts Γ for us. This section recasts many ideas of the previous sections using Agda notation, and introduces some new ideas. In particular, the 'type of types' `Type` is now cast as a hierarchy of types which can contain types at a 'smaller' level: One writes $\texttt{Set}_i$ to denote the type of types at *level* $i : \mathbb{N}$. This is a technical subtlety and may be ignored; instead treating every occurrence of $\texttt{Set}_i$ as an alias for `Type`.



Organisation Commentary. Since Agda is a DTL, it makes sense to begin with Π and DTs since one would expect them to occur everywhere else in a DTL —the motivation for things, such as Π, is in section 2.3. After Π, only may reasonable wonder about Σ since their close relationship was pointed out in section 2.3, Σ is not next on the tour since Agda records are syntactic sugar for data declarations having one constructor, so we need to discuss **data** after Π. Okay, we show 'data' and make use of the DTs already introduced; what's next? We show a concrete example of an ADT, namely '≡' since it will be used later on in examples in Chapter 5. Now that we're comfy with ADTs, we can go to ones with a single constructor, **record**s. But wait, Agda records behave like Agda modules, so let's talk about Agda **module**s first. After that, we can finally get to records (Σ-types) and we can do so very briefly since their underlying module/ADT nature has already been explained. In the next sibling section, 5.1, we show the interdefinability of packaging notions using Agda's syntactic sugar.

---

[54] 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

[55] Conor McBride. "Dependently typed functional programs and their proofs". PhD thesis. University of Edinburgh, UK, 2000. URL: http://hdl.handle.net/1842/374

[56] 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

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

Unicode Notation

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

2.4.1. Dependent Functions — Π-types

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

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

The Identity Function

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

id₁ id₂ id₃ : (X : Set) → X → X

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

All these functions explicitly require the type `X` when we use them, which is unfortunate 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.

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

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

Π Notation in Agda

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

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

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

2.4.2. Dependent Datatypes — ADTs

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

Informal Grammar Notation

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

Agda Rendition of Grammars

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

While the Agda form looks more verbose, it allows more possibilities that are difficult to express in the informal notation —such as, having *parameterised*[58] languages/types for which the constructors make words belonging to a *particular* parameter only; the `Vec` example below demonstrates this idea.

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

Interpreting the Door Values as Options

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

What is a constructor? A grammar defines a language consisting of sentences built from primitive words; a *constructor* is just a word and a word's *meaning* is determined by how it is used —c.f., `walk` above and the `Vec` construction below which gives us a way to talk

[58] With the "types as languages" view, one may treat a "parameterised type" as a "language with dialects". For instance, instead of a single language `Arabic`, one may have a *family of languages* `Arabic ℓ` that depend on a location ℓ. Then, some words/constructors may be accessible in *any* dialect ℓ, whereas other words can only be expressed in a *particular* dialect. More concretely, we may declare `SalamunAlaykum : ∀ {ℓ} → Arabic ℓ` since the usual greeting "hello" (lit. "peace be upon you") is understandable by all Arabic speakers, whereas we may declare `ShakoMako : Arabic Iraq` since this question form "how are you" (lit. "what is your colour") is specific to the Iraqi Arabic dialect.

about lists. The important thing is that a grammar defines languages, via words, without reference to meaning. Programmatically, constructors could be implemented as "(value position, payload data)"; i.e., pairs `(i, args)` where `i` is the position of the constructor in the list of constructors and `args` is a tuple values that it takes; for instance, `Door`'s constructors could be implemented as `(0,())`, `(1, ())`, `(2, (n))` for `Open`, `Closed`, `Ajar n` where we use `()` to denote "the empty tuple of arguments". The **purpose** of such types is that we have a number of *distinct* scenarios that may contain a 'payload' of additional information about the scenario; it is preferable to have **informative** (typed) names such as `Open` instead of strange-looking pairs `(0, ())`. In case it is not yet clear, unlike functions, a value construction such as `Ajar 10` cannot be simplified any further; just as the pair value `(2, 5)` cannot be simplified any further. Table 2.1 below showcases how many ideas arise from grammars.

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

Many useful ideas arise as grammars

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

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; in particular, the parameter values need to be the same in all constructor result type.

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 $\texttt{Set}\omega$.

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

Catenation is a $++ \longrightarrow +$ Homomorphism

```
infixr 40 _++_

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

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

Extended Commentary on Proof Trees: In section 2.2, we discussed how terms and trees coincide, but when focusing on *proof trees* the relationship gives us more. For instance, *the introduction and elimination rules of a type of trees correspond to the constructors and destructor of the type's grammar (ADT).*

Let me try to clarify what it means to say that “syntactic proof is an alternative to exhaustive case analysis” (valuations).

Solutions to families of problems can be phrased using names that can be defined using sets and functions between them; this is a *denotational semantics:* One solves a problem by looking up the definitions, denotations, of the names. In contrast, using ADTs provides a **proof system** to a problem: To solve a problem, one merely considers the “shape” of the problem to identify which rule (ADT constructor) to apply and continue this process recursively. That is, ADT proof systems generally provide a guidance to finding solutions. E.g., a propositional logic formula can be shown to be valid by showing every *valuation* (an assignment of values to variables) of its variables results in `true` — i.e., one must produce a function that takes in an arbitrary valuation and returns a proof of equality that the application of the valuation to the formula is `true` —; in contrast **natural deduction** is a collection of rules and one proves a formula is valid by constructing a tree whose conclusion is that formula; moreover, the shape of the formula usually determines (or, guides the construction of) the tree.

That is, ADTs give us a notion of proof that avoids checking all possible values for the variables. The ADT we design usually has its constructors —i.e., proof rules— to be sensible to the kind of problems we're interested in. This property is usually built-into the datatype; it is known as **soundness**: *The ADT only allows us to prove (i.e., form things) sensible with our intended interpretation; i.e., provable things are true.* The contrapositive —viz *non-true statements are not provable*— allow us to stop searching for a proof if we can find a counterexample. The converse —viz *true statements are provable; i.e., constructible via the ADT*— is called **completeness** and it is not as *practical* since one usually designs an ADT for a particular kind of problem —with a constrained amount of operations— rather than all kind of problems. Moreover, even if a true statement is provable, it may require an absurd amount of time to prove —e.g., the Ackermann function always terminates, and calling it on, say, $(4, 2)$ still has it terminating but long after I have died; or, more realistically, Agda will run out of resources and crash. By the same reasoning, typechecking in a DTL involves performing arbitrary computations, such as the Ackermann function, and so DTL type-checkers are not (practically) complete —however, in practice this is not an issue.

2.4.3. ADT Example: Propositional Equality

In this section, we present a notion of equality as an algebraic data type. Equality is a notoriously difficult concept, even posing it is non-trivial: "When are two things equal?" sounds absurd, since the question speaks about two things and two different things cannot be the same one thing. *Equality, whatever it means,*[59] *is about ignoring certain 'uninteresting' properties*;[60] below is a short hierarchy of 'sameness' with examples on ℕatural numbers.

1. **Syntactic equality:** "$l = r$" is true whenever l and r are literally the same string of symbols. E.g., $2 = 2$, or `suc suc zero = suc suc zero` .

   This is sometimes known as *intentional equality*; the equality of two expressions is 'built-in' the expressions themselves.[61]

2. **Definitional/judgemental equality:**[62] "$l = r$" is true whenever one looking-up definitions and applying them leads to syntactic equality. E.g., `suc zero + suc zero = suc suc zero`; i.e., $1 + 1 = 2$.

   Definitional equality is generally the form of equality taught at schools: Two expressions are equal if they both simplify, as much as possible, to the same thing. However, this approach —of '$=$' as an alias for a reflexive transitive reduction relation that permits a notion of 'simplification' or 'computation'— emphasises *operational behaviour* rather than *properties* of equality.

   This is also known as "normal form equality": One simplifies the two expressions, using definitions, until the two are syntactically indistinguishable. (The *normal form* of an expression is the most direct way of writing it; i.e., it consists of only constructors.) That is to say, definitional equality is the equivalence closure of a reduction relation —namely, the evaluation scheme of the programming language. In classical mathematics, this appears in the form of "semantic equality": Two things are equal when the values they *denote* coincide; e.g., "$2 + 2$" and "4" are clearly different, the first consisting of 3 symbols and the latter of 1 symbol, but after evaluation they denote the same value and so are treated equal. This is sometimes known as *extensional equality*; [46, 62, 78] .

3. **"≡" Propositional equality:** "$l = r$" is true exactly when one must perform some sort sort of case analysis of variables (i.e., induction) to arrive at a definitional equality.
   E.g., `suc m + suc zero = suc suc m`.

   In Agda, as shown below, the typing *judgement* `refl : 1 ≡`

[59] An **equivalence relation** _≈_ is a relationship that models *similarity*, thereby generalising the idea of equality. For _≈_ to be called "similarity, equivalence", it should satisfy:

1. (**Reflexivity**) "Everything is similar to itself"; i.e., $x \approx x$ is true for all x
2. "Similarity is a mutual relationship"
3. "Similarity is a transitive relationship"

[60] An informal code of conduct among mathematicians is that *interesting properties should be invariant under equivalence* —otherwise, they are 'evil' properties and should be used with caution. That is, for any interesting property P, one must have $P\,x = P\,y$ whenever x and y are "the same" —whatever that means. Working with equivalence-invariant properties is tantamount to working with an interface, a specfication, rather than a particular implementation.

[61] Pedantically, 2 is not the same as 2 viz "$2 = 2$", since the *actual occurrences* occupy different physical locations in this sentence.

[62] In classical mathematics, operators are usually *specified* by a set of laws —or universally characterised in category theory— and so users work with a specification rather than any particular set of definitional clauses. For instance, Cartesian products can be defined in numerous ways, but their universal mapping property suffices to characterise them and is powerful enough as an interface. In computing, the *choice of implementation* can make certain problems easier, more efficient, than others. For instnace, a stack of digits having at most n positions can be implemented by an array of length n or, much more efficiently, as an integer of n digits: Either approach gives us the stack interface —the methods `push, pop, insert, isempty`— but one is much more efficient.

`r` expresses that l and r as *judgmentally* (definitionaly) equal; i.e., a particular *term* is what signifies the equality as definitional. The equality that *can* be mentioned solely at the type level, and so *reasoned about*, is propositional equality: Two expressions, l and r, are propositionally equal, `∀ {x} → l ≡ r`, exactly when *any instantiation of the free variables*, x, results in definitionaly equal terms.

It is important to remember: "syntactic ⊆ definitional ⊆ propositional equality". The next kind of equality below, is orthogonal.

4. **Setoids / groupoids / equivalence relations:** "$l \approx r$" is proven using the assumption that `_≈_ : τ → τ → Set` is an equivalence relation and any properties of the type τ.

   For instance:

   a) **Extensionality:** $f \doteq g$ is proven for two *functions* by showing that $f\,x = g\,x$ for all appropriate arguments x.[63]
   "Extensionality is essential for abstraction"; i.e., functions are abstractions determined only by their input-output relationships —this is not true in computing, where efficiency is important and one speaks of algorithmic complexity. [55, 2]

   b) **Isomorphism:** $A \cong B$ is proven by exhibiting a non-lossy protocol between the two *types* A and B.[64]

   Extended Commentary: Experience has shown that, in Agda at least, the use of explicit equivalence relations is preferable to the use of propositional equality —i.e., `_≡_` is generally too strong, coarse, and one must generally use a finer equivalence relation. More generally, the use of setoids is the move from 'global identity types' `(A, _≡_)` to 'locally-defined identity types' `(A, _≈_)`, and more generally is *the move from sets to groupoids*: Two things are 'equal' exactly when there is a (necessarily invertible) morphism between them. Since Agda is constructive [11], its setoids could just as well have been called groupoids.

As a middle-ground, the *propositional equality datatype* is defined as follows. For a type `A` and an element `x` of `A`, we define the family of types/proofs of "being equal to x" by declaring only one inhabitant at index `x`.

Propositional Equality

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

[46] Jaakko Hintikka and Merrill B. Hintikka. "On Denoting what?" In: *The Logic of Epistemology and the Epistemology of Logic: Selected Essays*. Dordrecht: Springer Netherlands, 1989, pp. 165–181. ISBN: 978-94-009-2647-9. DOI: 10.1007/978-94-009-2647-9_11

[62] Gideon Makin. "Making sense of 'on denoting'". In: *Synth.* 102.3 (1995), pp. 383–412. DOI: 10.1007/BF01064122

[78] Bertrand Russell. "On Denoting". In: *Mind* XIV.4 (Jan. 1905), pp. 479–493. ISSN: 0026-4423. DOI: 10.1093/mind/XIV.4.479. eprint: https://academic.oup.com/mind/article-pdf/XIV/4/479/9872659/479.pdf

63 In classical maths, extensionality is equal to equality: "$\doteq\ =\ =$".

[55] Anne Kaldewaij. *Programming - the derivation of algorithms*. Prentice Hall international series in computer science. Prentice Hall, 1990. ISBN: 978-0-13-204108-9

[2] Andreas Abel and Gabriel Scherer. "On Irrelevance and Algorithmic Equality in Predicative Type Theory". In: *Log. Methods Comput. Sci.* 8.1 (2012). DOI: 10.2168/LMCS-8(1:29)2012

64 HoTT's univalence axiom, [83], says "isomorphism is isomorphic to equality" ("$\cong\ =\ =$") i.e., if two types are *essentially indistinguishable ('≅')* then we might as well treat them as *indistinguishable ('≡')*; which is what classical mathematicians do; compare with function extensionality. HoTT's univalence axiom wonderfully induces the expected definition of equality that one actually finds useful; e.g., categories are equal when they are equivalent.

[11] Andrej Bauer. "Five stages of accepting constructive mathematics". In: *Bulletin of the American Mathematical Society* (2016). DOI: https://doi.org/10.1090/bull/1556

This states that `refl {x}` is a proof of `l ≡ r` whenever `l` and `r` simplify, by definition chasing only, to `x` —i.e., both `l` and `r` have `x` as their normal form. This definition makes it easy to prove Leibniz's substitutivity rule, "equals for equals":

Transport along proofs

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

-- Alternatively...
cong : {A B : Set} {l r : A} (f : A → B)
     → l ≡ r  →  f l ≡ f r
cong refl = refl
```

How does `subst` work? An element of `l ≡ r` must be of the form `refl {x}` for some canonical form `x`; but if `l` and `r` are both `x`, then `P l` and `P r` are the *same type*. Pattern matching on a proof of `l ≡ r` gave us information about the rest of the program's type. By the same reasoning, we can prove that equality is the smallest possible reflexive relation. [65]

The Leibniz rule —*equals-for-equals:* `∀ {x y} → x ≡ y → f x ≡ f y` for any function f— is perhaps the most useful principle of equality. In Agda, if we know $x \equiv y$ by definitional (which includes syntactic) equality, then $f\,x \equiv f\,y$ is true *silently, automatically*: [66] Without ceremony, we can interchange one with the other. However, if `p : x ≡ y` is a propositional equality, then `cong f p : f x ≡ f y` ; i.e., we need to invoke the *particular* proof `p` in order to obtain the new proof. Finally, for setoid equivalence relations, one needs to prove the theorem `f-cong : ∀ {x y} → x ≈ y → f x ≈ f y` on a case-by-case basis, for each `f` one is interested in —think 'sets quotiented by an equivalence'. *Definitionally equal terms can be interchanged anywhere, silently,* and it is this property that makes them so remarkable.

Unsurprisingly, the Leibniz rule is so useful that it is sometimes used as *the* definition[67] of equality, also known as "The Law of Indiscernibles": `x ≡ y` exactly when, for every property `P` we have `P x ≡ P y` —two things are indistinguishable exactly when they share the same properties. That is, general equality is reduced to an already accepted notion of equality of truth values. One half of this rule is `cong`, proved above, but we do not have the other half: The extensionality principle `(∀ P → P x ≡ P y) → x ≡ y` is no longer true —indeed, observationally indistinguishable expressions are not necessarily equal since they may have been constructed (in syntactically) differently (ways). This is not surprising since propositional equality is intentional —after all it is based on syntactic indistinguishability.

[65] Any relation $\mathcal{R}$ that relates things to themselves —such that $x\,\mathcal{R}\,x$ for any x— must necessarily contain the propositional equality relation; i.e., $\_ \equiv \_ \subseteq \mathcal{R}$.

```
module _ {X} (_ℛ_ : X → X → Set)
  where

  ℛ-contains-≡ : Set
  ℛ-contains-≡
   = ∀ {x y}  →  x ≡ y  →  x ℛ y

  ℛ-reflexive : Set
  ℛ-reflexive  =  ∀ {x} → x ℛ x

  lrr :  ℛ-reflexive
      → ℛ-contains-≡
  lrr reflᵣ refl = reflᵣ

  lrr˘ :  ℛ-contains-≡
       → ℛ-reflexive
  lrr˘ go = go refl
```

"*$\mathcal{R}$ is reflexive precively when it contains $\_ \equiv \_$*" follows from (lrr) and (lrr˘), and is sometimes "the" definition of reflexivity.

[66] That is, if `refl : x ≡ y`, then, we can apply the definition of `cong`, to obtain `refl : f x ≡ f y`. That is, `cong refl` normalises to `refl`; whereas `cong p` cannot normalise since the definition of `cong` requires its argument to be the shape `refl` before any normalisation can occur. Hence, arbitrary propositional equality proofs `p : x ≡ y` lead to expression `cong f p : f x ≡ f y` which *can only* simplify in the same cases that allow `p` to simplify to `refl`.

There's only one constructor for equalities, so isn't every equality proof just `refl`? 'For the most part', *yes* —for more, see HoTT [83]. However, an arbitrary term `p : l ≡ r` is a *witness* that (1) both computations `l` and `r` terminate, and (2) they have the same normal form; and the definition of `cong` only works, computes, when we actually have `refl` in hand, so the issue becomes a matter of when can reduction happen.

[67] To wit:

$$x = y \quad = \quad (\forall P \;\bullet\; P\,x = P\,y)$$

The three occurrences of '=' above are all different!

One way [86, 5] to regain extensionality is to start with unityped terms *then* to define a type $\mathcal{T}$ to be partial-equivalence relation —i.e., an equivalence lacking reflexivity—, *then* define judgements $t : \mathcal{T}$ to mean $t\,\mathcal{T}\,t$; then every type is/comes with an equivalence relation and so one speaks of "equality at a type" with $l = r : \mathcal{T}$ meaning $l\,\mathcal{T}\,r$.

[86] Ed Voermans. "Pers as Types, Inductive Types and Types with Laws". In: *Declarative Programming, Sasbachwalden 1991, PHOENIX Seminar and Workshop on Declarative Programming, Sasbachwalden, Black Forest, Germany, 18-22 November 1991.* 1991, pp. 274–291. DOI: 10.1007/978-1-4471-3794-8_18

[5] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational equality, now!" In: *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007.* 2007, pp. 57–68. DOI: 10.1145/1292597.1292608

Is the `_≡_` datatype really equality? The name is definitely biased; below we change the names.

Discrete graphs with only self-loops

```
data _⟶_ {Node : Set} : Node → Node → Set
  where
    loop : {x : Node} → (x ⟶ x)
```

Instead of '≡' we have the long arrow '⟶', instead of `A` we have named the type parameter `Node`, and `refl` became `loop`. We may *interpret* the given type `Node` as a bunch of dots on a sheet of paper and a term `a : x ⟶ y` as an arc, arrow, from the dot named x to the dot named y. Whether we use this graphical[68] interpretation or the equality one is up to us, the users: The datatype itself carries no one, fixed, semantics. The change in perspective can offer great dividends; for instance, specialising the notion of a surjective graph homomorphism to this particular graph yields the observation that, in general, *injective means surjective on equations*:[69] For every proof `q : f x ≡ f y` there is a proof `p : x ≡ y` such that `cong f p ≡ q`.

As a slightly concrete example, if we define[70] addition on the natural numbers inductively on the first argument —i.e., `0 + n = n` and `suc m + n = suc (m + n)`— then one can show that 0 is the left identity of addition very *quickly* but to show that it is a right identity means we need to perform case analysis (i.e., induction) in order to make any progress (viz invoking the definition of $+$). We have two proofs of equality but one has a *shorter* proof length than the other: `0 + n ≡ n` is `refl` *immediately*, whereas `n + 0 ≡ n` *becomes* `refl` *after* performing n reduction steps to get into normal form.

In summary, 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.

[68] This is the groupoids interpretation! Moreover, whether we intepret the datatype as a *proposition* (equality) or as a *datastructure* (graph) is an example of the **propositions-as-types** interpretation used in DTLs.

[69] A more general definition of surjectivity can be seen in Baez ad Shulman's Lectures in n-categories and Cohomology [9].

[70] There are multiple, equivalent, definitions of addition; but we actually have to write one down in order to use it; and then this particular one is given special status by the programming language: The particular defining clauses are automatically theorems of addition (having zero-length proofs). More concretely, for our example, `0 + n` and `n` are indistinguishable to Agda, and so we can freely use such an identity law silently without mention; but the other identity law `n + 0 = n` requires explicit mention!. For instance, if `xs : Vec A (0 + n)` then `xs : Vec A n`; but if `xs : Vec A (n + 0)` then `subst p _ : Vec A n` where we must ceremonially transport `xs`, 'coerce', along the proof `p : n + 0 ≡ n`. This issue pops up in the wild in useful, simple, programs such as the catenation of vectors; try it!

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 v = subst +-sym v
                  -- where +-sym : ∀ {n m}  →  m + n ≡ n + m
```

That is, whenever one has a proof `p : l ≡ r`, if `p` is the `refl` *constructor*, then *l and r are equal by definition chasing*; otherwise, they require a 'non-trivial' proof and are thus *propositionally equal*. In particular, to type check `refl : f(x) = y` for some function `f`, the system must actually perform the computation `f` on input `x` *then* check for syntactic equality against `y`. Hence, equalities may contain non-trivial computational content and typechecking may involve non-trivial computational effort; e.g., $\mathsf{refl} : 2^{100} \equiv 2^{100}$ takes some 'time' to typecheck.

2.4.4. Modules —Namespace Management; $\Pi\Sigma$-types

For now, Agda modules are not first-class[71] constructs and essentially only serve to delimit (possibly parameterised) namespaces, thereby avoiding name clashes —as such, there are only a few associated keywords, which we show briefly in this section. The use of modules is exemplified by the following snippets.

A Simple Module

```
module A where

  𝒩 : Set
  𝒩 = ℕ

  private
    x : ℕ
    x = 3

  y : 𝒩
  y = x + 1
```

Using It

```
use₀ : A.𝒩
use₀ = A.y

use₁ : ℕ
use₁ = y
  where open A

open A

use₂ : ℕ
use₂ = y
```

Parameterised Modules

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

Name = Function

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

Using Them

```
use'₀ : ℕ
use'₀ = B.y 3

module C = B 3

use : ℕ
use = C.y

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

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

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

Module combinators supported in the current implementation of Agda

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

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

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

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

2.4.5. Records — Σ-types

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

`record` $\approx$ `module` + `data` with one constructor

The class of types along with a value picked out

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

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

Defining Instances

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

ex₁ : PointedSet
ex₁ = MkIt ℕ 3

open PointedSet

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

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

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

Simple Uses

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

use¹ : ℕ
use¹ = point
    where open PointedSet ex₀

open PointedSet

use² : ℕ
use² = blind ex₀ true
```

Pattern Matching on Records

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

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

use⁴ (MkIt C x)
  = x
```

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

So much for records.[72]

[72] Agda `record`s are particular ADTs/`data`, which have been discussed in detail up to this point. They have `module`-like beavhiour, which has also been discussed, and so, reasonably, the discussion on records is terse.

3. Motivating the problem —Examples from the Wild

In this chapter, we motivate the problems —for which we will[0] find solutions for— by finding examples within public libraries of code developed in 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. Encouraged by program correctness activities, our focus will inexorably lead to embedding program specifications at the *type level*, but we will see that *sometimes* it is more pragmatic to relocate the specification to the *value level* (section 3.1); this then leads to choosing more apt names (section 3.2) and to mixing-in features to an existing module (sections 3.1.3, 3.3, 3.4). To illustrate the core concepts, we will use the , algebraic structures Magma Semigroup, and Monoid [1] .

Incidentally, the common solutions to the problems presented may be construed as **design patterns for dependently-typed programming**. Design patterns[5] are algorithms[2] yearning to be formalised. The power of the host language dictates whether design patterns remain as informal directions to be implemented in an ad-hoc basis then checked by other humans, or as a library methods that are written once and may be freely applied by users. For instance, the Agda `Algebra.Morphism.Structures` "library"[3] presents *only* examples of the homomorphism design pattern —which shows how to form operation-preserving functions for a few chosen algebraic structures. Examples, 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.

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

0 This chapter lays out the problems; there's nothing "new" here *besides* collecting existing problems in DTLs and the current ways they are handled by DTL practitioners.

1 A *magma* `(C, ⨾)` is a set `C` and a binary operation `_⨾_ : C → C → C` on it; a *semigroup* is a magma whose operation is associative, `∀ x, y, z • (x ⨾ y) ⨾ z = x ⨾ (y ⨾ z)`; and a *monoid* is a semigroup that has a point `Id : C` acting as the identity of the binary operation: `∀ x • x ⨾ Id = x = Id ⨾ x`. For example, real numbers with subtraction `(ℝ, -)` are only a magma whereas numbers with addition `(ℝ, _+_, 0)` form a monoid. The *canonical models* of magma, semigroup, and monoid are trees (with branching), non-empty lists (with catenation), and possibly empty lists, respectively — these are discussed again in section 7.5.

5 Definition: *A general, reusable solution to a commonly occurring problem.*

2 Definition: *A finite sequence of instructions to be followed to accomplish a goal.*

3 All references to the Agda Standard Library refer to the current version 1.3. The library can be accessed at `https://github.com/agda/agda-stdlib`.

Chapter Contents

3.1. Simplifying Programs by Exposing Invariants at the Type Level

In this section, we want to discuss how "unbundled (possibly value-parameterised) presentations" can be used to simplify programs and statements about elements of shared types. We begin with a ubqutious problem[4] that happens in practice: Given a list `[`x_0`, `x_1`, …, `x_{n-1}`]`, how do we get the k^{th} element of the list? Unless $0 \leq k < n$, we will have an error. The issue is clearly at the 'bounds', 0 and n, and so, for brevity, we focus on the problem of extracting the first element of a list —i.e., the first bound. The resulting unbundling solution has its own problems, so afterward, we consider how to phrase composition of programs in general and abstract that to phrasing distributivity laws. Finally, from the previous two discussions, we conclude with a promising suggestion that may improve library design.

In particular, this section is about "how a user may wish things were bundled" and a suggestion to "how a library designer should bundle data".

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

The purpose of this section is to demonstrate the related, yet different, ideas below.
(1) Isomorphism is not indistinguishable from equality.
(2) Propositional equality is not equal to definitional equality.
(3) Equivalent presentations are not equivalent in different, real usage scenarios.
The first two subsections here are concrete instances of the more general situation and reader familiar with DTLs is encouraged to skip ahead to section 3.1.3.

3.1.1. Avoiding "Out-of-bounds" Errors

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

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

Lists as Algebraic Data Types

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

Then, to define `head l` for any list `l`, we consider the *possible shapes* of the variable list `l`. The two possible shapes are an empty list `[]` and a prepending of an element `x` to another list `xs`. In the second case, the the list has `x` as the first element and so we yield that. Unfortunately, in the scenario of an empty list, there is no first element to return! However, `head` is typed `List A → A` and so it must somehow produce an `A` value from any given `List A` value. In general, this is not possible: If `A` is an empty type, having no values at all, then `[]` is the only possible list of `A`'s, and so `head []` is a value of `A`, which contradicts the fact that `A` is empty. Hence, either `head` remains a partially-defined[6] function or one has to "add fictitious elements to every type"[7] such as $\mathsf{undefined}_A : A$. However, in a DTL, we can *add the non-emptiness condition* `l ≠ []` to the type level and have it *checked at compile-time by the machine rather than by the user.*

Trying to define the `head` function.

Partially defined `head`

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

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

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

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

Non-emptiness Predicate

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

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

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

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

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

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

Exposing Information At the Type Level

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

Our type of vectors[8] is defined intentionally using the same constructor names as that of lists, which Agda allows. Notice that the first constructor is declared to be a member of the type `Vec A 0`, whereas the second declares `x :: xs` to be in `Vec A (suc n)` when `xs` is in `Vec A n`, and so `l : Vec A n` implies that the length of `l` is `n`. In particular, if `l : Vec A (suc n)` then `l` has a positive length and so is non-empty; i.e., non-emptiness can be expressed directly in the type of `l`.

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

Non-emptiness proviso at the type level

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

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

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

Simple Partial Specfications of List Operations

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

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

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

In theory, lists and vectors are the same[9] —where the latter are essentially lists indexed by their lengths. In practice, however, the additional length information stated up-front as an integral part of the data structure makes it not only easier to write programs that would otherwise be awkward or impossible[10] in the latter case. For instance, above we demonstrated that the function `head`, which extracts the first element of a non-empty list, not only has a difficult

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

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

type to read, but also requires an auxiliary relation/type in order to be expressed. In contrast, the vector variant has a much simpler type with the non-emptiness proviso expressed by requesting a positive length.

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

Equivalent structures, but different usability profiles.

Eek!

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

3.1.2. "To Bundle or Not To Bundle": Structure vs Predicate Style Presentations

Given two different structures that share some sub-component, expressing that sharing post-facto can be very cumbersome, while if the sharing is expressed via parameters, things are simple —even though both encodings are equivalent. (This is 'essentially' the same problem as discussed in the previous section but in a different guise, as a stepping stone to the more general situation.)

The phenomenon of exposing attributes at the type level to gain flexibility applies not only to derived concepts such as non-emptiness, but also to explicit features of a datatype. A common scenario is when two instances of an algebraic structure share the same carrier and thus it is reasonable to connect the two somehow by a coherence axiom. But for such an equation to be well-typed, we need to *know* that the composition operators work on the *same kind* of programs phrases —it is surprisingly not enough to know that each combines certain kinds of program phrases that happen to be the same kind.

That is, the "same problem" arises when, for example, discussing the interaction between sequential program composition `_⨾_` and parallel program composition `_||_`: The *simultaneous* execution of programs `P`-then-`P'` and `Q`-then-`Q'` results in the same behaviour as the *sequential* execution of `P`-and-simultaneously-`Q` then `P'`-and-simultaneously-`Q'`. That is, `(P ⨾ P') || (Q ⨾ Q') = (P || Q) ⨾ (P' ⨾ Q')`.

Consider what is perhaps the most popular instance of structure-sharing known to many from childhood, in the setting of rings: We have an additive structure `(R, +)` and a multiplicative structure `(R, ×)` on the same underlying set `R`, and their interaction is dictated by distributivity axioms, such as $a \times (b + c) = (a \times b) + (a \times c)$. As with `head` above, depending on which features of the structure are exposed upfront, such axioms[11] may be either difficult to express or relatively easy. Below are the two possible ways to present a structure admiting

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

[11] "Obviously sharing the same type" requires 'do-nothing' conversion functions!

a type and a binary operation on that type.

To bundle or to not bundle?

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

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

A $\texttt{Magma}_0$ **is a** pair ⟨C, op⟩ of a type C and an operation op on that type!

A $\texttt{Magma}_1$ **on** a given type C **is a** one-tuple ⟨op⟩ consisting of a binary operation on that type!

Magma₀ ≅ (Σ C : Set • Magma₁ C)

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

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

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

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

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

In **theory**, parameterised structures are no different from their unparameterised, or “bundled”, counterparts. Indeed, we can easily prove `Magma₀ ≅ (Σ C : Set • Magma₁ C)` by “packing away the parameters” and `∀ (C : Set) → Magma₁ C ≅ (Σ M : Magma₀ • M.Carrier ≡ C)` by “abstracting a field as if it were a parameter” — this is known as ‘Σ-padding’. Like the first isomorphism[12] , the second is proven just as easily but suffers from excess noise introduced by the Σ-padding, namely extra phrases “ `, refl` ” that serve to keep track of important facts, but are otherwise unhelpful. The proofs generalise easily on a case-by-case basis to other kinds of structures, but they cannot be proven internally to Agda in full generality.

Let us consider[13] *using* the first presentation. When structures “pack away” all their features, the simple distributivity property becomes a bit of a challenge to write and to read.

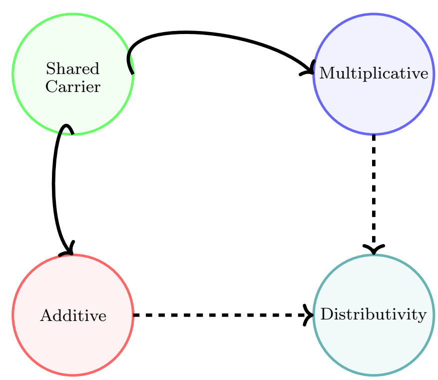
[13] A discussion of propositional equality versus equality-by-construction can be found in section .

Distributivity is Difficult to Express

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

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

  field shared-carrier :  R₊ ≡ R×

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

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

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

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

Bundled forms require (curved) coercisions



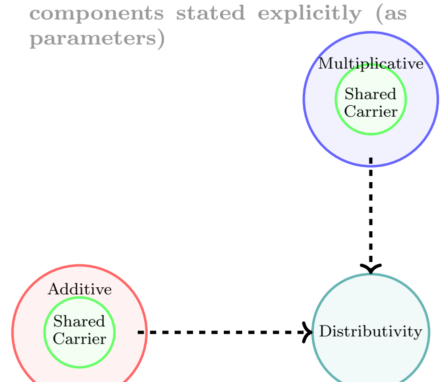
Again, in theory, parameterised structures are no different from their unparameterised, or "bundled", counterparts. However, in **practice**, even when multiple presentations of an idea are *equivalent* in some sense, there may be specfic presentations that are *useful* for particular purposes[14] . That is, in a depeendely-typed language, equivalence of structures and their usability profiles do not necessairly go hand-in-hand. Indeed, below we can phrase the distributivity axiom nearly as it was stated informally earlier since the shared carrier is declared upfront.

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

Distributivity is Expressed Easily with Unbundled Structures

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

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

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

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

Unbundled forms have shared components stated explicitly (as parameters)



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

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

Unbundling Design Pattern

If a feature of a class is shared among instances, then use an unbundled form of the class to avoid "coercion hell". See Sections 3.1.3, 4.1, 7.2.

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

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

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

If we refer to the former as Is$\mathcal{X}$ and the latter as $\mathcal{X}$, then we can see similar instances in the standard library for $\mathcal{X}$ being:

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

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

```
record IsSemigroup {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                    (· : Op₂ A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    isEquivalence : IsEquivalence ≈
    assoc         : Associative ·
    ·-cong        : · Preserves₂ ≈ ⟶ ≈ ⟶ ≈
```

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

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

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

Predicate Design Pattern

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

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

Since the standard library uses the predicate pattern, `Is`$\mathcal{X}$, which requires all sets and function symbols, the Σ-padding anti-pattern becomes a necessary evil. Instead, it would be preferable to have the family $\mathcal{X}_i$ which is the same as `Is`$\mathcal{X}$ but only[15] takes i-many elements —c.f., $\mathtt{Magma}_0$ and $\mathtt{Magma}_1$ above. However, writing these variations and the necessary functions to move between them is not only tedious but also error prone. Later on, also demonstrated in [45], we shall show how the bundled form $\mathcal{X}$ acts as ***the*** definition, with other forms being derived-as-needed.

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

[15] 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`.

[45] 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

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

Typeclass Design Pattern

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

As discussed in section 5.1, when there are multiple instance of an $\mathcal{X}$-structure on a particular type, only one of them may be marked for instance search in a given scope.

Type Classes for Mathematics in Type Theory [79] discusses the numerous problems of bundled presentations as well as the issues of unbundled presentations and settles on using typeclasses along with their tremendously useful instance search mechanism. Since we view $\mathcal{X}_1$ as a particular choice in the family $(\mathcal{X}_w)_{w \in \mathbb{N}}$, our approach is to instead have library designers define $\mathcal{X}_0$ and let users *easily, mechanically, declaratively,* produce $\mathcal{X}_w$ for any 'parameterisation waist' $w : \mathbb{N}$. This idea is implemented for Agda, as an in-language library, and discussed in chapter 7.

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

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

3.2. Renaming

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

$x(f)$ Using x as a *function* and f as an *argument*.; likewise $\frac{\partial x}{\partial f}$.

With the exception of discussions involving the Yoneda Lemma, or continuations, such a notation is simply *'wrong'*.

$a \times a = a$ An idempotent operation denoted by multiplication; likewise for commutative operations.

It is more common to use addition or join, '$\sqcup$', to denote idempotent operations.

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

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

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

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

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

Renaming Design Pattern

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

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

1. Agda's "standard library" [4],
2. The "RATH-Agda" library [53], and
3. A recent "agda-categories" library [52].

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

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

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

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

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

the Agda definitions and domain to use the library.

◇ Consequently, there are many inconsistencies in naming.

The `open` ⋯ `public` ⋯ `renaming` ⋯ pattern shown below will be reappear later, section 6.3, as a library method.

The “Shape” of Renaming Blocks in Agda

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

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

Keep an eye out for all those `renaming` (η_1 `to` η_1`'; ...;` η_k `to` η_k`'`) lines!

3.2.1. Renaming Problems from Agda’s Standard Library

Below are four excerpts from Agda’s standard library, notice how the prototypical notation for monoids is renamed **repeatedly** *as needed*. Sometimes it is relabelled with additive symbols, other times with multiplicative symbols.

Additive Renaming —IsNearSemiring

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

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

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

Additive Renaming Again —IsSemiringWithoutOne

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

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

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

Please keep a lookout for the `renaming ( ⋯ )` lines; it is such a *schematic shape* that is important —not the actual content![16]

[16] Whence the intentionally tiny font!

Additive Renaming a 3^{rd} Time and Multiplicative Renaming —IsSemiringWithoutAnnihilatingZero

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

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

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

Additive Renaming a 4^{th} Time and Second Multiplicative Renaming —IsRing

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

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

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

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

Original —Prototypical— Notations

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

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

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

Renaming Helper Modules

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

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

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

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

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

More Necessary Renaming Helper Modules

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

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

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

3.2.2. Renaming Problems from the RATH-Agda Library

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

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

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

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

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

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

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

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

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

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

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

Each `Setoid`$\mathcal{XXX}$ takes around 10 lines, for a total of roughly 600 lines!

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

3.2.3. Renaming Problems from the Agda-categories Library

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

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

- Functors have fields named `F₀`, `F₁`, `F-resp-≈`, ...
  - This could be considered reasonable even if one has a functor named `G`.

More meaningful names may be `obj`, `mor`, `mor-cong` —which refer to a functor's "obj"ect map, "mor"phism map, and the fact that the "mor"phism map is a "cong"ruence.

- Such lack of concern for naming might be acceptable for well-known concepts such as functors, where some communities use $\mathtt{F}_i$ to denote the object/0-cell or morphism/1-cell operations. However, considering subcategories one sees field names `U`, `R`, `Rid`, `_∘R_` which are wholly unhelpful.

Instead, more meaningful names such as `embed`, `keep`, `id-kept`, `keep-resp-∘` could have been used.

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

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

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

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

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

Symbol Soup

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

Here are a few downsides of not renaming:

1. The type of the function is difficult to comprehend; though it need not be.

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

2. The short proof is difficult to read!

   The repeated terms such as `η β B` and `η β A` could have been renamed with mnemoic-names such as η_1, η_2 or η_s, η_t.

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

Declare `_≈₀_` and `_≈₁_` to be `Setoid._≈_ (F₀ Nat[Hom[C][-,c],F] (F , A))` and, respectively, `Setoid._≈_ (F₀ F B)` .

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

$f : \mathsf{source}\ f \to \mathsf{target}\ f$
or $f : X_1 \to X_2$.

⋆ ⋆ ⋆

Just an application of a functor's morphism mapping.

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

3.3. Redundancy, Derived Features, and Feature Exclusion

A tenet of software development is not to over-engineer solutions. For example, if we need a notion of untyped composition, we may use `Monoid`. However, at a later stage, we may realise that units are inappropriate[17] and so we need to drop them to obtain the weaker notion of `Semigroup`. In weaker languages, we could continue to use the monoid interface at the cost of "throwing an exception" whenever the identity is used. However, this breaks the *Interface Segregation Principle: Users should not be forced to bother with features they are not interested in* [64]. A prototypical scenario is exposing an expressive interface, possibly with redundancies, to users, but providing a minimal self-contained counterpart by dropping some features for the sake of efficiency or to act as a "smart constructor" that takes the least amount of data to reconstruct the rich interface. Tersely put: One axiomatisation may be ideal for verifying instances, whereas an equivalent but possibly longer axiomatisation may be more amicable for calculation and computation.

[17] 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.

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

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

In particular, the Category type and the natural isomorphism type are in-

Redundancy can lead to silently used equalities

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

-- Notice:  assocˡ ≈ sym assocʳ

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

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

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

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

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

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

Incidentally, since this is not a library method, inconsistencies[18] are bound to arise. Such issues could be reduced, if not avoided, if library methods could have been used instead of manually implementing design patterns.

[18] In particular, in the $\mathcal{X}$ and $\mathcal{X}$`Helper` naming scheme: The `NaturalIsomorphism` type has `NIHelper` as its minimised version, and the type of symmetric monoidal categories is oddly called `Symmetric'` with its helper named `Symmetric`.

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

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 }
```

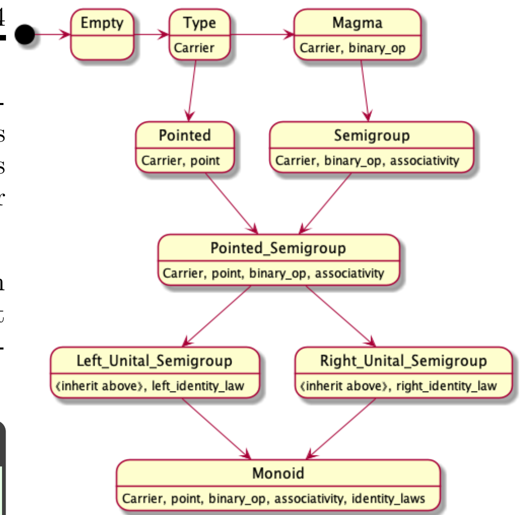
Another example of purposefully keeping redundant features is for the sake of efficiency; e.g., quoting RATH-Agda (section 15.13), *For division semi-allegories, even though right residuals, restricted residuals, and symmetric quotients all can be derived from left residuals, we still assume them all as primitive here, since this produces more readable goals, and also makes connecting to optimised implementations easier.* For instance, the above semigroup type could have been augmented with an ordering if we view `_⨾_` as a meet-operation. Instead, we could lift such a derived operation as a primitive field, in case the user has a better implementation.

Efficient Redundancy Design Pattern (RATH-Agda section 17.1)

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

3.4. Extensions

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



Possible hierarchy leading to Monoid

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

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

Extending Semigroup to Obtain Monoid

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

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

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

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

open Monoid  -- (⋆, *)

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

Extensions are not flattened inheritance

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

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

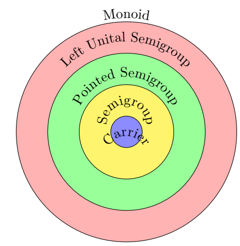
This nesting scenario happens rather often, in one guise or another. The amount of syntactic noise required to produce a simple instantiation is unreasonable: *One should not be forced to work through the hierarchy if it provides no immediate benefit.*

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

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

Extensions require deep —'staircase'— projections

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



Extension Design Pattern

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

Extension Design Pattern Prototype

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

While library designers may be content to build `Monoid` out of `Semigroup`, users should not be forced to learn about how the hierarchy was built. Even worse, when the library designers decide to incorporate, say, `RightUnitalSemigroup` instead of the left unital form, then all users' code would break.

Instead, it would be preferable to have a 'flattened' presentation for the users that "does not leak out implementation details". That is, a 'flattened' hierarchy may be *seen* as a single package, consisting of the fields throughout the hierarchy, possibly with default implementations, yet still be able to view the resulting package at base levels in the hierarchy —c.f., section 3.3. Another benefit of this approach is that it allows users to utilise the package without consideration of how the hierarchy was formed, thereby providing library designers with the freedom to alter it in the future.

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

3.5. Conclusion

After ‘library spelunking’, we are now in a position to summarise the problems encountered, when using existing[19] 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.

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

3.5.1. Lessons Learned

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

Example (1). A large number of proofs can be discharged by merely pattern matching on variables —this works since the case analysis reduces the proof goal into a trivial reflexitivity obligation, for example. The number of cases can quickly grow thereby taking up space, which is unfortunate since the proof has very little to offer besides verifying the claim. In such cases, a pre-process, perhaps an “editor tactic”, could be utilised to produce the proof in an auxiliary file, and reference it in the current file.

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

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

“By hand” is tedious, error prone, and obscures the generic rewriting method!

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

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

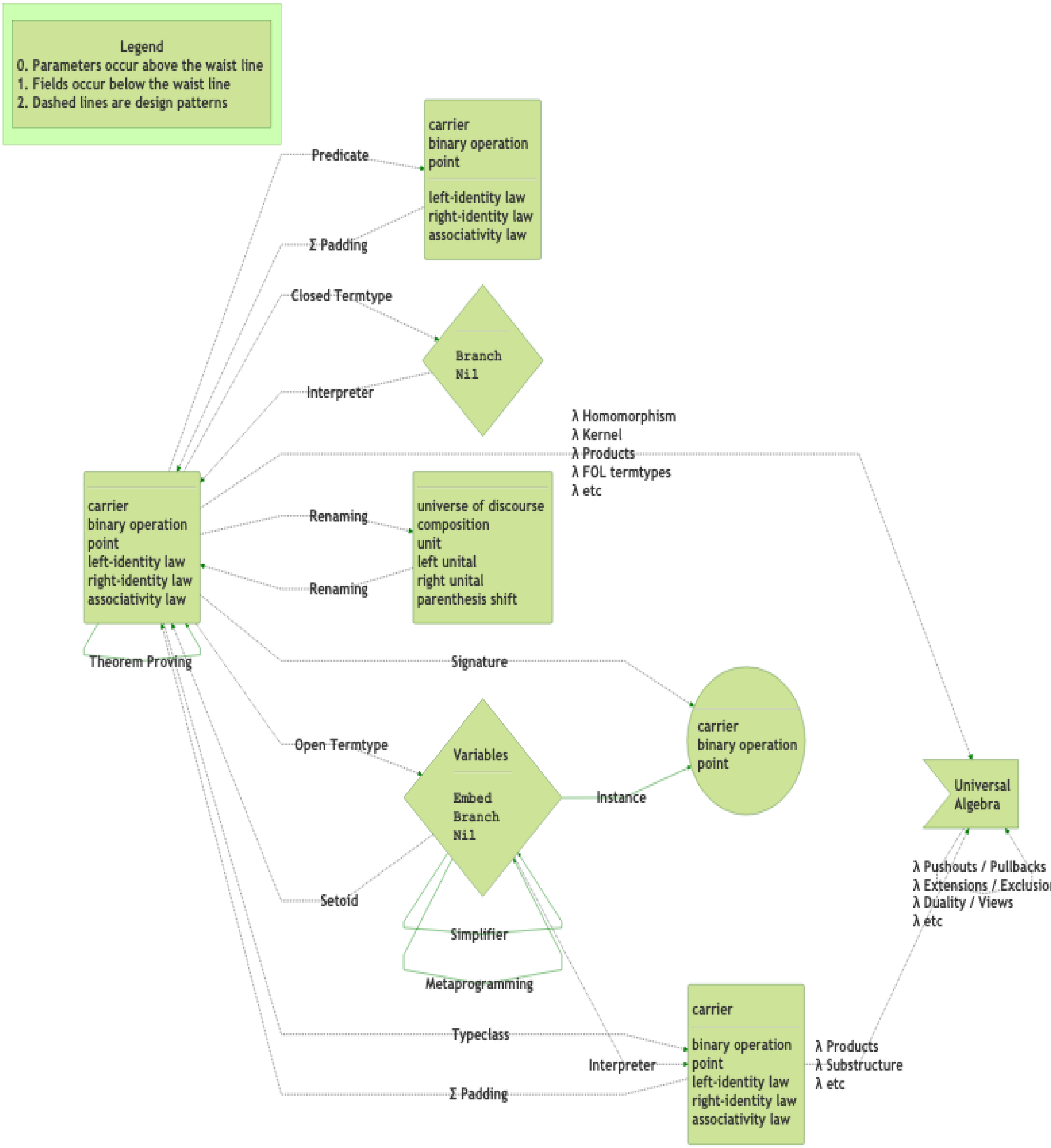


Figure 3.1.: PL Research is about getting free stuff: From the left-most node, we can get a lot!

3.5.2. One-Item Checklist for a Candidate Solution

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

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

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

4. Contributions of the Thesis

With the necessary background covered in Chapter 2 and motivating examples discussed in Chapter 3, we are in a position to discuss the contributions of this thesis in a technical fashion. The first section discusses the primary problem the thesis aims to address. The second section outlines the objectives of this thesis and discusses the methodology used to achieve those objectives. The third, and final, section discusses the outcomes of the thesis effort.

Since 'grammars' and 'algebraic datatypes' are just $\mathcal{W}$ell-founded tress, we abbreviate such terms to '$\mathcal{W}$-types'. Technically, every inductive datatype is expressible as a $\mathcal{W}$-type —a discussion we leave for Chapter 5.

Chapter Contents

4.1. Problem Statement

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

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

This is understandable: Function symbols generally vary more often than proof obligations. (This is discussed in detail in Section 3.1.3 and rendered in concrete Agda code in Section 7.2.) However, when users do not yet have the necessary parameters $\mathtt{p}_i$, they need to use a curried (or *bundled*) form of the module and so library developers also provide a module $\mathcal{X}$ which packs up the parameters as necessary fields within the module; i.e., $\mathcal{X}$ has the shape $\Pi^0\Sigma$ by "pushing down" the parameters into the record body. Unfortunately, there is a whole spectrum of modules $\mathcal{X}_w$ that is missing: These are the module $\mathcal{X}$ where only w-many of the original parameters are exposed with the remaining being packed-away into the module body; i.e., having the shape $\Pi^w\Sigma$ for $0 \leq w \leq n$ —in subsequent chapters, we refer to w as "the waist" of a package former. It is tedious and error-prone to form all the $\mathcal{X}_w$ by hand; such 'unbundling' should be mechanically achievable from the completely bundled form $\mathcal{X}$. A similar issue happens when one wants to *describe a computation* using module $\mathcal{X}$, then its function symbols need to have associated syntactic counterparts —i.e., we want to interpret $\mathcal{X}$ as a $\mathcal{W}$-type instead of a $\Pi^n\Sigma$-type —; the tedium is then compounded if one considers the family $\mathcal{X}_w$. Finally, instead of combinations of Π,Σ,$\mathcal{W}$, a user may need to treat a module $\mathcal{X}$ as an arbitrary container type[21]; in which case, they will likely have to create it by hand.

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

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

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

[21] Thorsten Altenkirch et al. "Indexed containers". In: *J. Funct. Program.* 25 (2015). DOI: 10.1017/S095679681500009X. URL: https://doi.org/10.1017/S095679681500009X

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[22] 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 produce the necessary morphisms to make this construction into a pushout. Likewise, we would expect products, unions, intersections, and substructures of theories —when possible, and then to be constructed by users. In this thesis, we only want to provide a fixed set of meta-primitives from which usual and (un)conventional notions of grouping may be defined.

Objective 3: Provide a Semantics

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

4.3. 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

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

3. Demonstrate that there is an expressive yet minimal set of module meta-primitives which allow common module constructions to be defined

4. Demonstrate that relationships between modules can also be *mechanically* generated.

   ◇ In particular, if module $\mathcal{B}$ is obtained by applying a user-defined 'variational' to module $\mathcal{A}$, then the user could also enrich the child module $\mathcal{B}$ with morphisms that describe its relationships to the parent module $\mathcal{A}$.

   ◇ E.g., if $\mathcal{B}$ is an extension of $\mathcal{A}$, then we may have a "forgetful mapping" that drops the new components; or if $\mathcal{B}$ is a 'minimal' rendition of the theory $\mathcal{A}$, then we have a "smart constructor" that forms the rich $\mathcal{A}$ by only asking the few $\mathcal{B}$ components of the user.

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

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

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

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

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

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

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

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

9. Finally, the resulting framework is *mostly type-theory agnostic*: The target setting is DTLs but we only assume the barebones as discussed in 7.6; if users drop parts of that theory, then *only* some parts of the framework will no longer apply.

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

5. A Π-Σ-$\mathcal{W}$ View of Packaging Systems

The thesis is that contexts serve as a unified notion of packaging.

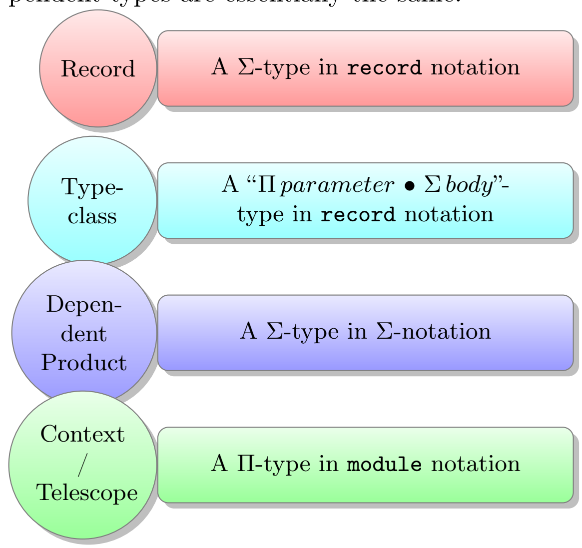
As such, in this chapter, in section 5.1, we demonstrate three possible ways to define monoids in Agda and argue their equivalence; thereby, showing that structuring mechanisms are in effect accomplishing the same goal in different ways: They package data along with a particular *usage interface*. As such, it is not unreasonable to seek out a unified notion of **package** —namely, contexts. After showing how the usual record formulation of monoids is equivalent to a pure contextual one, in section 5.2 we verify that contexts are indeed promising by discussing how other dependently-typed languages (DTLs) view contexts and signatures. In particular, we compare the construction of a tiny graph library in Coq with its alternative form in Agda: Coq's modules *are* contexts whose declarations are either marker `Parameter` or as `Axiom` —an equivalent form of the contexts we have in mind. Unlike Coq, we want to use the contexts for algebraic datatypes as well. As such, we review $\mathcal{W}$-types in section 5.3. Finally, in section 5.4, we formalise our approach for contexts serving as a generic packaging mechanism. The formalism is in dependent type theory, whereas the next chapter provides a Lisp implementation and the chapter after that shows an Agda implementation.

Chapter Contents

5.1. 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.



5.1.1. Three Ways to Define Monoids

A **monoid** is a collection, say `Carrier`, along with an operation, say `_⨾_`, on it and a chosen point, say `Id`, from that collection. **Monoids model composition:** We have a bunch of things called `Carrier` —such as programs or words—, we have a way to 'mix' or 'compose' two things `x` and `y` to get a third `x ⨾ y` —such as forming a big program from smaller pieces or a story from words— which has an selected 'empty' thing that does not affect composition —such as the do-nothing program or the 'empty word' which does not add content to a story. There are three *typical* ways to formalise the type of monoids: (1) As a `record` since a monoid is a bunch of things together; (2) as a 'typeclass' (parameterised record) since we want to *specialise* the carrier dynamically or to have instance search (which is an invaluable feature in, for example, Haskell, which organises its libraries using typeclasses and instance search); (3) as a raw unsugared Σ-type since we want to explicitly disallow the inherent `module`-nature of Agda's `record`s. A DTL allows for redundancies like this so users can solve their problems in ways they see best.

The type of monoids is formalised below as `Monoid-Record`; additionally, we have the derived result: `Id`-entity can be popped-in and out as desired.

Monoids as Agda Records —The usual mathematical definition

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

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

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

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

Instance Resolution: The double curly-braces `{{...}}` serve to indicate that the given argument is to be found by *instance resolution*. For example, if we declare `it : {{e : A}} → B`, then `it` is a `B`-value that is formed using an `A`-value; but which `A`-value? Unlike a function which requires the `A`-value as input, `it` will "look up" an `A`-value in the list of names that are marked for look-up by the keyword `instance`. If multiple `A`-values are marked for look-up, it is not clear which one should be used; as such, *at most one*[24] value can be provided for lookup and this value is called "the declared `A`-instance", whence the name 'instance resolution'. Recall that Agda records automatically come with an associated module, and so the `open` clause, above, makes the name `pop-Id-Rec : {{M : Monoid-Record}} → (x y : Monoid-Record.Carrier M) → ...` accessible; in-particular, this name uses instance resolution: The derived result, `pop-Id-Rec`, can be invoked without having to mention a monoid, provided a unique `Monoid-Record` value is declared for instance search —otherwise one must use named instances [25]. We will return to actually declaring and using instances in the next section.

A value of `Monoid-Record` is essentially a tuple `record{Carrier = C; ...}`; so the carrier is *bundled at the value level.* If we to speak of "monoids with the specific carrier $\mathcal{X}$", we need to *bundle the carrier at the type level.* This is akin to finding the carrier "dynamically, at runtime" versus finding it "statically, at typechecking time". [26]

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

[25] Wolfram Kahl and Jan Scheffczyk. "Named Instances for Haskell Type Classes". In: 2001

[26] An accessible introduction to semantics and typeclasses, using a monoid of functions as the running example,

Monoids as 'Typeclasses'/'Generics' —Parameterisation on the underlying set

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

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

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

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

Monoids as Dependent Sums —Using none of Agda's built-in syntactic sugar

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

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

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

can be found in: Elliott. "Denotational design with type class morphisms". In: 2016. URL: http://conal.net/papers/type-class-morphisms/type-class-morphisms-long.pdf.

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[27] type, we could have declared a monoid to be a semigroup with additional pieces:

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

There are a large number of hyper-graphs indicating how monoidal interfaces could be built from their parts, we have only presented a stratified view for brevity. In particular, `Monoid-Σ` is the extreme unbundled version, whereas `Monoid-Record` is the other extreme, and there is a large spectrum in between —all of which are somehow isomorphic[28]; e.g., `Monoid-Record` ≅ `Σ C : Set • MonoidOn C`. Our envisioned system would be able to derive any such view at will[29] and so programs may be written according to one view, but easily repurposed for other view with little human intervention.

5.1.2. 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.

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

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

[29] 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

Instance Declarations

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

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

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

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

No Monoids Mentioned at Use Sites

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

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

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

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

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

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

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

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

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

5.1.3. A Fourth Definition —Contexts

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

Monoids as Telescopes

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

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

"Squint and They're The Same:" Notice that this is nothing more than the named fields of `Monoid-Record` but not[30] bundled. Additionally, if we insert a Σ before each name we

[30]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.

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 on to. -}
record-from-telescope : Monoid-Record
record-from-telescope
  = record { Carrier = Carrier
           ; Id      = Id
           ; _⨾_     = _⨾_
           ; lid     = lid
           ; rid     = rid
           ; assoc   = assoc
           }
```

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

Agda

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

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

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

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

Agda

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

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

Agda

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

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

Demonstrating the interdefinablity of different notions of packaging cements our thesis that it is essentially *utility* that distinguishes packages more than anything else —just as `data` language's words (constructors) have their meanings determined by *utility*. Consequently, explicit distinctions have lead to a duplication of work where the same structure is formalised using different notions of packaging. In chapter 6 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.

5.2. Contexts are Promising

The current implementation of the Agda language[31,32] has a notion of second-class modules which may contain sub-modules along with declarations and definitions of first-class citizens. The intimate relationship between records and modules is perhaps best exemplified here since the current implementation provides a declaration to construe a record as if it were a module —as demonstrated in the previous section. This observation is not specific to Agda, which is herein only used as a presentation language. Indeed, other DTLs (dependently-typed languages) reassure our hypothesis; the existence of a unified notion of package:

◇ **The centrality of contexts**

The **Beluga** language has the distinctive feature of direct support for first-class contexts[33]. A term `t(x)` may have free variables and so whether it is well-formed, or what its type could be, depends on the types of its free variables, necessitating one to either declare

[31] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda — A Functional Language with Dependent Types". In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17–20, 2009. Proceedings.* 2009, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6

[32] Ulf Norell. "Towards a Practical Programming Language Based on Dependent Type Theory". See also http://wiki.portal.chalmers.se/agda/pmwiki.php. PhD thesis. Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology, Sept. 2007

[33] Brigitte Pientka. "Beluga: Programming with Dependent Types, Contextual Data, and Contexts". In: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings.* 2010, pp. 1–12. DOI: 10.1007/978-3-642-12251-4_1. URL: https://doi.org/10.1007/978-3-642-12251-4%5C_1

them before hand or to write, in Beluga, `[ x : T |- t(x) ]` for example. As argued in the previous section, contexts are essentially dependent sums. In contrast to Beluga, **Isabelle** is a full-featured language and logical framework that also provides support for named contexts in the form of 'locales'[34,35]; unfortunately it is not a dependently-typed language.

◇ **Signatures as an underlying formalism**

Twelf[36] is a logic programming language implementing Edinburgh's Logical Framework[37,38,39] and has been used to prove safety properties of 'real languages' such as SML. A notable practical module system[40] for Twelf has been implemented using signatures and signature morphisms.

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

The current implementation of **Coq**[41,42] provides a "copy and paste" operation for modules using the `include` keyword. Consequently it provides a number of module combinators, such as `<+` which is the infix form of module inclusion[43]. Since Coq module types are essentially contexts, the module type `X <+ Y <+ Z` is really the catenation of contexts,

[34] Clemens Ballarin. "Locales and Locale Expressions in Isabelle/Isar". In: *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*. 2003, pp. 34–50. DOI: 10.1007/978-3-540-24849-1_3. URL: https://doi.org/10.1007/978-3-540-24849-1%5C_3

[35] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. "Locales - A Sectioning Concept for Isabelle". In: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*. 1999, pp. 149–166. DOI: 10.1007/3-540-48256-3_11. URL: https://doi.org/10.1007/3-540-48256-3%5C_11

[36] Frank Pfenning and The Twelf Team. *The Twelf Project*. 2015. URL: http://twelf.org/wiki/Main_Page (visited on 10/19/2018)

[37] Christian Urban, James Cheney, and Stefan Berghofer. *Mechanizing the Metatheory of LF*. 2008. arXiv: 0804.1667v3 [cs.LO]

[38] Florian Rabe. "Representing Isabelle in LF". in: *Electronic Proceedings in Theoretical Computer Science* 34 (Sept. 2010), pp. 85–99. ISSN: 2075-2180. DOI: 10.4204/eptcs.34.8. URL: http://dx.doi.org/10.4204/EPTCS.34.8

[39] Aaron Stump and David L. Dill. "Faster Proof Checking in the Edinburgh Logical Framework". In: *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*. 2002, pp. 392–407. DOI: 10.1007/3-540-45620-1_32. URL: https://doi.org/10.1007/3-540-45620-1%5C_32

[40] 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: https://doi.org/10.1145/1577824.1577831

[41] Christine Paulin-Mohring. "The Calculus of Inductive Definitions and its Implementation: the Coq Proof Assistant". In: invited tutorial

[42] Jason Gross, Adam Chlipala, and David I. Spivak. *Experience Implementing a Performant Category-Theory Library in Coq*. 2014. arXiv: 1401.7694v2 [math.CT]

[43] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*. Apr. 2018. DOI: 10.5281/zenodo.1219885. URL: https://hal.inria.fr/hal-01954564

where later items may depend on former items. The **Maude**[44,45] framework contains a similar yet more comprehensive algebra of modules and how they work with Maude theories.

◇ **Parameters of records are actually their fields**

The **Arend** proof assistant [77, 50] does not distinguish between record parameters and record fields (as such, fields can be *specialised* dynamically; i.e., Π and λ are essentially identified, but we will form a combinator '$\Pi{\to}\lambda$' in Chapter 5). This is the exact insight that we arrived at, [45], independently at around the same time that the first version of Arend was released.

Arend provides a *built-in* solution, whereas we show how such a solution to the unbundling problem can be formed as a reflection library in a DTL. Moreover, our target setting is for both proving *and* programming.

Arend is based on intuitionistic logic, like Agda, but is otherwise intended for theorem proving in homotopy type theory[46].

It is important to consider other languages so as to how see their communities treat module systems and what uses cases they are interested in: *It is important to draw wisdom from many different places; if you take it from only one place, it becomes rigid and stale*[47]. In the next section, we shall see a glimpse of how the Coq community works with packages, and, to make the discussion accessible, we shall provide Agda translations of Coq code.

5.2.1. Coq Modules as Generalised Signatures

Module Systems parameterise programs, proofs, and tactics over structures. In this section, we shall form a library of simple graphs[48] to showcase how Coq's approach to packages is

[44] Manuel Clavel et al., eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-71940-3. DOI: 10.1007/978-3-540-71999-1. URL: https://doi.org/10.1007/978-3-540-71999-1

[45] Francisco Durán and José Meseguer. "Maude's module algebra". In: *Sci. Comput. Program.* 66.2 (2007), pp. 125–153. DOI: 10.1016/j.scico.2006.07.002. URL: https://doi.org/10.1016/j.scico.2006.07.002

[46] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013

[47] Michael Dante DiMartino and Bryan Konietzko. *Avatar, the last airbender*. Premiered on Nickelodeon. 2005

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

essentially in the same spirit[49] as the proposed definition of generalised signatures: A sequence of name-type-definition tuples where the definition may be omitted. To make the Coq accessible to readers, we will provide an Agda translation that only uses the `record` construct in Agda —completely ignoring the `data` and `module` forms which would otherwise be more natural in certain scenarios below— in order to demonstrate that *all packaging concepts essentially coincide in a DTL.*

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

In Coq, a `Module Type` contains the signature of the abstract structure to work from; it lists the `Parameter` and `Axiom` values we want to use, possibly along with notation declaration to make the syntax easier. (The naming in the following module, `Graph`, is slightly inappropriate since connectedness is generally via paths not edges —which are chosen for brevity.)

Graphs —Coq

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

  (* Obtain convenient syntactic sugar. *)
  Infix "<=" := Edges : order_scope.
  Open Scope order_scope.

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

Graphs —Agda

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

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

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

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

Birds' Eye View

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

Booleans are Graphs —Coq

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

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

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

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

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

Booleans are Graphs —Agda

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

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

Minimisation as a function on modules —Coq

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

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

Type of module 'Min'

```
Module Type (G : Graph).
  Import G.
  Definition min a b : Vertex := if (decidable a b) then a else b.
  Parameter case_analysis: forall P : Vertex -> Type, forall x y,
        (x <= y -> P x) -> (y <= x -> P y) -> P (min x y).
End Min.
```

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

Minimisation as a function on modules —Agda

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

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

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

open Min
```

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

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

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

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

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

```
Conjunction = Min BoolGraph

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

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

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

A module-to-module function —Coq

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

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

A module-to-module function —Agda

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

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

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

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

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

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

  max : Vertex → Vertex → Vertex
  max = min I

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

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

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

Signatures and structures in Coq, Agda, and JavaScript

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

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

Modules and module types only differ in intended utility

5.3. ADTs as $\mathcal{W}$-types

Earlier in the chapter we demonstrated the interdefinability of various structuring mechanisms; yet, there are times when it would be prudent to have the *syntax* of a concept (such as monoid) in hand so as to, say, generate terms of that type or to simplify them as follows.

A syntax for 𝕄onoid terms

```
-- 𝕄onoid terms over carrier C
data 𝕄 (C : Set) : Set where
  inj : C → 𝕄 C
  Id : 𝕄 C
  _⨾_ : 𝕄 C → 𝕄 C → 𝕄 C

-- If x and y are in simplified form, then so is x ⨾′ y
_⨾′_ : {C : Set} → 𝕄 C → 𝕄 C → 𝕄 C
Id ⨾′ y = y
x ⨾′ Id = x
x ⨾′ y  = x ⨾ y

-- Discard as much Id's as possible
simplify : {C : Set} → 𝕄 C → 𝕄 C
simplify (a ⨾ b) = simplify a  ⨾′ simplify b
simplify it = it

popId : ∀ {C} {x y  : 𝕄 C}   →   simplify (x ⨾ (Id ⨾ y)) ≡  simplify (x ⨾ y)
popId = refl
```

Records and typeclasses have a similar shape — as quantifiers $\mathcal{Q}x : A \bullet B\,x$ — and if we want to interpret contexts as grammars, we should use a similar shape as well. Discussing such a shape is the goal of this section.

5.3.1. When does `data` actually define a type?

Grammars, **data** declarations, *describe* the *smallest* language that has the constructors as words. What if no such language exists? Indeed, not all grammars are 'sensible' in that they define a language. For instance, `One` below is a language of only **one word**, `MakeOne`; whereas `None` is a language with **no words**, since to form a phrase `MakeNone n` first requires we form `n`, which leads to infinite regress, and so there are no *finite* words. Even worse, `What` describes no language at all —whcih Agda barks as being *not strictly positive*.

Describing Possibly Non-Existent Languages

```
data One : Set where
  MakeOne : One

data None : Set where
  MakeNone : None → None

data What : Set where
 MakeWhat : (What → What) → What
```

Recall that $(A \to B) = \Pi \_ : A \bullet B$ and, when A is finite , $(A \to B) \cong (\Pi \_ : A \bullet B) \cong B^{|A|}$. As such, a function `What → What`, above, is `What`-many arguments, each of type `What`. But how many arguments is that exactly? We need to actually know the type `What`, which is the type being defined. As such, the above **data** does not actually define any type.

5.3.2. $\mathcal{W}$

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

Important Observation

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

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

Notice that we have again encountered the problem of a syntax that is "too powerful" for the concepts it denotes: We can declare grammars (ADTs) that do not describe *any* language. Since a grammar consists of a number of *disjoint* ("Σ") constructor clauses that take a *tuple* ("Π") of arguments, it suffices to consider when "polynomial"[50] descriptions
`F X = (Σ a : A • Π b : B a • X)` actually describe a language. That is, when is there a function $\mathbf{F}\,X \to X$ and what is the *smallest* X with such a function? The values of $\mathbf{F}\,X$ are pairs (a, f) where $a : A$ and $f : B\,a \to X$; so we may take the collection of *only* such pairs to be the language described by $\mathbf{F}$, and it is thus the smallest such collection. This language is called a **$\mathcal{W}$-type**.

Descriptions of Languages That Necessarily Exist

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

$\mathcal{W}$-types in Agda

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

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

Categorically speaking, polynomial functors —i.e., type formers of the shape `F X = Σ a : A • Π b : B a • X`, "sums of products" or a "disjoint union of possible constructors and their arguments"— have "initial algebras" named `W = (𝒲 a : A • B a)`, which are the smallest

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

languages described by $\mathbf{F}$. That is, $\mathcal{W}$-types are the initial algebras of polynomial functors; that is, $\mathbf{F}$ has an initial algebra `sup : F W → W`.
Moreover, every strictly positive type operator can be expressed in the same shape as $\mathbf{F}$ and so they all have an initial algebra —for details see [51]. Inductive families arise as indexed $\mathcal{W}$-types which are initial algebras for dependent polynomial functors, and [52] have shown them to be constructible from non-dependent ones in locally cartesian closed categories. That is, indexed $\mathcal{W}$-types can be obtained from ordinary $\mathcal{W}$-types. See also [53].

5.3.3. $\mathcal{W}$-types generalise trees

To further understand $\mathcal{W}$-types —and to justify the name sup!—, consider the type `Rose A` of "multi-branching trees with leaves from A". *$\mathcal{W}$-types generalise the idea of rose trees:* Each list of children trees `xs : List (Rose A)` can be equivalently[54] replaced by a *tabulation* `cs : Fin (length xs) → Rose A` that tells the i-th child of `xs`. That is, **$\mathcal{W}$-types are trees with branching degrees** $(B\,a)_{a:A}$.

Rose trees

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

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

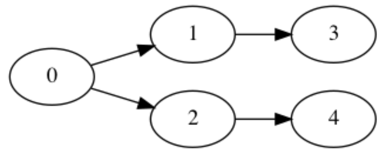
The `example` tree is shown diagrammatically below.

[51] Peter Dybjer. "Representing inductively defined sets by wellorderings in Martin-Löf's type theory". In: *Theoretical Computer Science* 176.1-2 (Apr. 1997), pp. 329–335. ISSN: 0304-3975. DOI: 10.1016/s0304-3975(96)00145-4. URL: http://dx.doi.org/10.1016/s0304-3975(96)00145-4

[52] Nicola Gambino and Martin Hyland. "Wellfounded Trees and Dependent Polynomial Functors". In: *Types for Proofs and Programs* (2004), pp. 210–225. ISSN: 1611-3349. DOI: 10.1007/978-3-540-24849-1_14. URL: http://dx.doi.org/10.1007/978-3-540-24849-1_14

[53] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. "Representing Nested Inductive Types Using W-Types". In: *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings.* Ed. by Josep D́ıaz et al. Vol. 3142. Lecture Notes in Computer Science. Springer, 2004, pp. 59–71. ISBN: 3-540-22849-7. DOI: 10.1007/978-3-540-27836-8_8. URL: https://doi.org/10.1007/978-3-540-27836-8%5C_8, The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study: https://homotopytypetheory.org/book, 2013, Jacopo Emmenegger. *W-types in setoids.* 2018. arXiv: 1809.02375v2 [math.LO]

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



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

Rose trees

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

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

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

The *height* of a tree, is an ordinal, and is defined to be the supremum[55] —i.e., the least upper bound— of the height of its elements:

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

This may be reason why the only constructor of $\mathcal{W}$-types is named `sup`. Indeed, we may interpret $\mathcal{W} a : A \bullet B\, a$ as the least upper bound of all languages (ordered by language inclusion) that contain terms " `f(args)` " where $f : A$ is a 'function symbol' and $args : B\, f$ is an 'appropriately-shaped argument' —e.g., concrete terms " $\mathtt{f(t_0,\ t_1,\ \ldots,\ t_n)}$ " are an instance of this idea, as witnessed by $\sup\ f\ \mathsf{child}_f$ with $\mathsf{child}_f : \mathsf{Fin}\,(\mathsf{length}\ f) \to \mathsf{Term}$ defined by $\mathsf{child}_f(i) = t_i$.

In contrast, `𝒲 a : A • Fin n` is a data type with `A`-many clauses that *each* make `n` recursive calls; this is an *empty type* since every construction requires `n` many existing constructions — however, it is still a type, unlike `Noo` above. That is[56], if `B a` is non-empty for all `a`, then $\mathcal{W}$

[55] The supremum of the empty set of natural numbers is, by definition, 0.

$$\sup \emptyset = 0$$

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

[56] A $\mathcal{W}$-type is empty precisely when it has no nullary constructor; see exercise 5.17 of [83].

`a : A • B a` is empty, since in order to form an element `sup a c`, we need to have defined before-hand `c(b) : (`$\mathcal{W}$ `a : A • B a)` for each one of the elements `b` of `B a`.

5.4. $\Pi\Sigma\mathcal{W}$ Semantics for Contexts

Parameterised Agda modules (and records) are contexts —i.e., Coq modules— that have all their parameters first then followed by only by named symbols that must have term definitions. These are a mixture of Π and Σ types: The parameters are captured by a Π type and the defined named are captured by Σ-types as in " Π `parameters` • Σ `body` ". (In general, since Coq modules allow parameters to occur *after* locally defined names, one could use `let`-clauses to mimic such an approach with Π-Σ-types.) *Were* it possible, dynamically on-the-fly, to only request a subsegement of the parameters list then we have a solution to the unbundling problem. Moreover, as we will show, contexts can also be furnished with $\mathcal{W}$ semantics to obtain a termtype for the structure being defined —this is one of our primary contributions.

A quick recap of how Π, Σ, $\mathcal{W}$ serve programming: "Π" Product types are the essence of structured data —all languages have some form of product type, such as *record, class, struct, object, JSON object, hashmap.* **"Σ"** Most data structures involve alternatives, choice, which is expressed by sum types —a value of a sum type is thus used ('eliminated') by case analysis. Perhaps the simplest example of a sum type is the type of truth values: Acting depends on whether a particular condition is true *or* false. The eliminator ("how to use the Boolean") for the Booleans is the familiar `if...then...else` construct. More generally, sum types may be used to define *finitely enumerated types*, the types whose values are of an explicitly declared set and whose elimination form ("how to use them") is case analysis. For instance, the cardinal directions —`Up, Down, Left, Right`— are an enumerated type that may be useful in an system requiring navigation, whereas the type `Maybe` τ ::= `Just` τ `|` `Nothing` models *pointers to values of type* τ. Notice that `Maybe` τ is an enumerated type where *its values may hold values of* τ —these are alternatives with a 'payload': Indeed, `Maybe` $\tau \cong \tau$ + $\mathbb{1}$. **"$\mathcal{W}$"** Next, one wonders, can we have an enumerated type whose values may involve other values of the type being defined: Enter inductive types; which are captured by $\mathcal{W}$-types.

For brevity we will work with the polymorphic lambda calculus, 'System F', whose terms are as follows —assuming a given set of variable $\mathsf{Name}s$.

Term	::=	x	(variable, an element of Name)
	|	$\lambda x \bullet e$	(lambda abstraction)
	|	$f\,e$	(application)
	|	$\Pi x : A \bullet B$	(dependent function type)
	|	Type_i	(ith universe; $i = 0, 1, 2, ...$)

We may then takes *types* to be the terms that *describe* other terms; as such, there is one grammar for both Rather than two grammars.

$\neg$(`$\mathcal{W}$ a : A • B a`) $\cong \neg$ (Σ `a : A •` $\neg$ `B a`)

Type constructions $T : \mathsf{Type} \to \mathsf{Type}$ give algebraic data types —"initial algebras"— I satisfying $I \cong T(I)$ by the definition $I \;=\; \Pi\, X : \mathsf{Type} \bullet (T\, X \to X) \to X$. We use this idea to regain the other useful type formers; e.g., for $\mathcal{W}$-types we have $T\, X \;=\; \Sigma\, a : A \bullet (B\, a \to X)$ and so $\mathcal{W}$-types are encoded as $\Pi X : \mathsf{Type} \bullet ((\Sigma\, a : A \bullet (B\, a \to X)) \to X)) \to X$, or equivalently —using Σ-elimination— as $\Pi\, X : \mathsf{Type} \bullet (\Pi\, a : A \bullet \Sigma\, f : B\, a \to X \bullet X) \to X$.

Type	Encoding
$\mathbb{0}$	$\Pi\, X : \mathsf{Type} \bullet X$
$\mathbb{1}$	$\Pi\, X : \mathsf{Type} \bullet (X \to X)$
$A \to B$	$\Pi\, \_ : A \bullet B$
$A \times B$	$\Pi\, i : \mathsf{Fin}\, 2 \bullet \lambda\{0 \to A, 1 \to B\}\, i$
$\Sigma\, x : A \bullet B$	$\Pi\, X : \mathsf{Type} \bullet (\Pi\, x : A \bullet \Pi\, y : B \bullet X) \to X$
$\mathcal{W}\, x : A \bullet B$	$\Pi\, X : \mathsf{Type} \bullet (\Pi\, x : A \bullet \Pi\, f : (B \to X) \bullet X) \to X$

Recall that our contexts are left-growing lists of variables annotated with their types. We use left-growing instead of the more common right-growing lists since we are working with *dependent* contexts: In the context $x : A; \Gamma$ we expect the name x to be available in the rest of the context Γ.

$$\begin{array}{lcll} \Gamma & ::= & . & \text{(empty context)} \\ & | & x : A, \Gamma & \text{(context extension)} \end{array}$$

We shall use the same notation — viz $x_0, x_1, \ldots, x_n$ and $\cdot$ — to denote lists and make use of a number of common list operations, including the following.

$$\begin{array}{lcl} \mathsf{foldr1} & : & \forall\{\tau\}(\_\oplus\_ : \tau \to \tau \to \tau) \to \mathsf{List}\,\tau \to \tau \\ \mathsf{foldr1}\ \_\oplus\_\ (x, \cdot) & = & x \\ \mathsf{foldr1}\ \_\oplus\_\ (x, xs) & = & x \oplus (\mathsf{foldr1}\ \_\oplus\_\ xs) \\ \\ \mathsf{any} & : & \forall\{\tau\}(p : \tau \to \mathbb{B}) \to \mathsf{List}\,\tau \to \mathbb{B} \\ \mathsf{any}\, p\, \cdot & = & \mathsf{false} \\ \mathsf{any}\, p\, (x, xs) & = & p\, x \ \vee\ \mathsf{any}\, p\, xs \end{array}$$

We can then define a number of semantics functions on contexts.

$$\begin{array}{lcl} \Pi[\![\_]\!] & : & \mathsf{Context} \to \mathsf{Term} \\ \Pi[\![\cdot]\!] & = & \mathbb{1} \\ \Pi[\![x : A, \Gamma]\!] & = & \Pi\, x : A \bullet \Pi[\![\Gamma]\!] \end{array}$$

For instance, $\Pi[\![$ `Carrier : Type, point : Carrier` $]\!]$ `=` Π `Carrier : Type` $\bullet$ Π `point : Carrier` $\bullet$ $\mathbb{1}$; the right-hand side is the uninteresting function sending its input to the only element of $\mathbb{1}$. We will find practical uses for this operation in conjunction with the others.

Of course any proper Π-*term* can be converted to a context:

$$\begin{array}{lcl} \Gamma[\![\_]\!] & : & \mathsf{Term} \to \mathsf{Context} \\ \Gamma[\![\Pi\, x : A \bullet B]\!] & = & x : A, \Gamma[\![B]\!] \\ \Gamma[\![t]\!] & = & \cdot \end{array}$$

By structural induction, one can verify $\Gamma\,[\![\,\Pi[\![c]\!]\,]\!] \;=\; c$ —but we do not, in general, have an isomorphism.

The next semantics function is hardly more complicated.

$$
\begin{array}{lcl}
\Sigma[\![\_]\!] & : & \mathsf{Context} \to \mathsf{Term} \\
\Sigma[\![\cdot]\!] & = & \mathbb{1} \\
\Sigma[\![x : A, \Gamma]\!] & = & \Sigma\, x : A \bullet \Sigma[\![\Gamma]\!]
\end{array}
$$

For instance, Σ ⟦`Carrier : Type, point : Carrier` ⟧ `=` Σ `Carrier : Type` $\bullet$ Σ `point : Carrier` $\bullet$ $\mathbb{1}$; the right-hand side is essentially a record type but lacking any syntactic sugar. This is the usual *record semantics* of contexts.

The next semantics function is perhaps the most complicated. Given a context Γ and an elected type name τ, this operation keeps only the names of Γ that 'hit' τ —i.e., they have types being functions targeting τ— then it 'd'rops that h'ead' —c.f., `dead` below— from the resulting types in the context *and* produces a 'hole' for any recursive call; finally, the resulting types are summed as well as the holes.

$$
\begin{array}{lcl}
\mathcal{W}[\![\_]\!] & : & \mathsf{Context} \to \mathsf{Name} \to \mathsf{Term} \\
\mathcal{W}[\![\Gamma]\!]\,\tau & = & \mathsf{if\ any}\ \ (\_\mathsf{hits}\,\tau)\ \Gamma \\
 & & \mathsf{then}\ \mathcal{W}\ (\mathsf{foldr1}\ +\ (\mathsf{dead}\,\tau\,\Gamma))\ (\mathsf{foldr1}\ \triangledown\ (\mathsf{holes}\,\tau\,\Gamma)) \\
 & & \mathsf{else}\ \mathbb{0}
\end{array}
$$

It is important that we use foldr1 and not foldr since we do not want to append a any type for the recursive base case (empty list) —otherwise, our ADTs would all have 'one more' new constructor. The '$+$' is the sum construction on types, whereas '$\triangledown$' is the sum selection operator —i.e., sum eliminator [57] For instance, $\mathcal{W}$ ⟦`Carrier : Type, point : Carrier` ⟧ `Carrier` `=` $\mathcal{W}$ $\mathbb{1}$ `(λ _ → \Fin 0)`; the right-hand side is essentially a `data` declaration with one ('$\mathbb{1}$') nullary ('Fin 0') constructor. This semantics, as far as we know, is novel.

The helper functions required to define $\mathcal{W}[\![\_]\!]$ include the standard textual substitution of terms, the subterm relation '$\subseteq$', and '$x\sharp t$' for the operation of the number of times a name x occurs in a term t. The two unmentioned operations below are the incidence relation '$\lhd$' and

[57] **cats:programming_with_bananas.**

the context subtraction operation '-'.

$$
\begin{array}{lcl}
\_ \lhd \_ & : & \mathsf{Name} \times \mathsf{Term} \to \mathsf{Context} \to \mathbb{B} \\
x : A \ \lhd \ \Gamma & = & \text{“typing } x : A \text{ draws from context } \Gamma\text{”} \\
x : A \ \lhd \ \cdot & = & \mathsf{false} \\
x : A \ \lhd \ (y : B, \Gamma) & = & x = y \ \vee \ y \subseteq A \ \vee \ x : A \lhd \Gamma
\end{array}
$$

$$
\begin{array}{lcl}
\_ - \_ & : & \mathsf{Context} \to \mathsf{Context} \to \mathsf{Context} \\
\cdot \ - \ \Gamma' & = & \cdot \\
(x : A, \Gamma) \ - \ \Gamma' & = & \mathsf{if}\ x : A \ \lhd \ \Gamma' \ \mathsf{then}\ \Gamma \ - \ (x : A, \Gamma') \ \mathsf{else}\ x : A, (\Gamma - \Gamma')
\end{array}
$$

$$
\begin{array}{lcl}
\_\mathsf{hits}\_ & : & \mathsf{Term} \to \mathsf{Name} \to \mathbb{B} \\
x \,\mathsf{hits}\, y & = & (x = y) \qquad \text{(Variable case)} \\
(\Pi x : A \bullet B) \,\mathsf{hits}\, y & = & B \,\mathsf{hits}\, y \qquad \text{(Π-type)} \\
t \,\mathsf{hits}\, y & = & \mathsf{false} \qquad \text{(All other cases)}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{dead} & : & \mathsf{Name} \to \mathsf{Context} \to \mathsf{List\ Term} \\
\mathsf{dead}\, A \cdot & = & \cdot \\
\mathsf{dead}\, A\, (y : B, \Gamma) & = & \mathsf{if}\ B \,\mathsf{hits}\, A \\
& & \mathsf{then}\ \Sigma[\![\mathsf{init}\,\Gamma[\![B]\!]]\!][A \coloneqq \mathbb{1}], \mathsf{dead}\, A\, \Gamma \\
& & \mathsf{else\ dead}\, A\, \Gamma \ - \ y : B
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{holes} & : & \mathsf{Name} \to \mathsf{Context} \to \mathsf{List\ Term} \\
\mathsf{holes}\, A \cdot & = & \mathsf{Fin}\, 0 \\
\mathsf{holes}\, A\, (y : B, \Gamma) & = & \mathsf{if}\ B \,\mathsf{hits}\, A \ \mathsf{then\ Fin}\, (A \sharp B - 1), \mathsf{holes}\, A\, \Gamma \ \mathsf{else\ holes}\, A\, (\Gamma \ - \ y : B)
\end{array}
$$

Rather than prove any correctness of these generic operations, we will, in Chapter 7, mechanise these them in Agda. The mechanisation is a non-trivial contribution since it is the "real-world details" where things become rather involved. For instance, unlike our supposed setup above, in Agda terms have a much larger syntax and so a number of combinators must be developed along the way —including, manipulation of De Bruijn indices. Moreover, the resulting Agda setup is pragmatic since it uses monadic `do`-notation to achieve a simple concrete syntax for contexts and, unlike the above setup, it is a library and not a 'proof-of-concept' development from scratch. "In theory, it's doable; actually doing it is another matter!"

Finally, there is a family of useful semantics combinators built on top of $\Pi[\![\_]\!]$. For any semantics function $\mathcal{Q}[\![\_]\!] : \mathsf{Context} \to \mathsf{Term}$, we have the family "$\Pi^w \mathcal{Q}$" for each *waist* $w : \mathbb{N}$.

$$
\begin{array}{lcl}
\Pi^w \mathcal{Q}[\![\_]\!] & : & \mathsf{Context} \to \mathsf{Term} \\
\Pi^0 \mathcal{Q}[\![\Gamma]\!] & = & \mathcal{Q}[\![\Gamma]\!] \\
\Pi^{w+1} \mathcal{Q}[\![\cdot]\!] & = & \mathcal{Q}[\![\cdot]\!] \\
\Pi^{w+1} \mathcal{Q}[\![x : A, \Gamma]\!] & = & \Pi\, x : A \bullet \Pi^w \mathcal{Q}[\![\Gamma]\!]
\end{array}
$$

For instance, the single context `Carrier : Type, Tree : Type, leaf : Carrier → Tree,` `branch : Tree → Tree → Tree` can be used to obtain a parameterised record and a parameterised datatype —*both being different useful ways to view the same context.* follows.

$\Pi^1\Sigma \approx$ Typeclass

```
  Π¹Σ ⟦ Carrier : Type, Tree : Type, leaf : Carrier → Tree
        , branch : Tree → Tree → Tree ⟧
=
  Π Carrier : Type • Σ Tree : Type • Σ leaf : Carrier → Tree
        • Σ branch : Tree → Tree → Tree • 𝟙
≈
  record CollectionOn (Carrier : Set) : Set₁ where
      field
        collection : Set -- ‘Tree’ above
        singleton  : Carrier → collection -- ‘leaf’ above
        merge      : collection → collection → collection -- ‘branch’ above
```

$\Pi^1\mathcal{W} \approx$ Termtypes

```
  Π¹𝒲 ⟦ Carrier : Type, Tree : Type, leaf : Carrier → Tree
        , branch : Tree → Tree → Tree ⟧ Tree
=
  Π Carrier : Type • 𝒲 (Carrier + 𝟙 × 𝟙) (λ{inl _ → Fin 0, inr _ → Fin 2})
≈
  data TreeOn (Carrier : Set) : Set where
      leaf   : Carrier → TreeOn Carrier
      branch : TreeOn Carrier → TreeOn Carrier → TreeOn Carrier
```

As the examples show sometimes after restructuring a context it can be useful to perform a *renaming* operation. The current implementation of Agda does not allow for the declaration of freshly named entities and so we relegate this aspect to the Emacs Lisp prototype of Chapter 6. The prototype is able to perform such renaming and *remember* the relationship [58] to the original datatype by augmenting the resulting `record` with coercions —‘forgetful operations’— to the original, parent, context. Consequently, the prototype —even though it is useful by itself— acts as a guide for features that would be ideal to implement in a DTL capable of supporting them as a library.

Finally, the “do-it-yourself” in the title of the thesis is that the resulting Agda library of Chapter 7 is designed around Π⟦_⟧ and Σ⟦_⟧ but users would use any other, possibly personal, semantics operation $\mathcal{Q}$⟦_⟧.

[58] William M. Farmer. “A New Style of Mathematical Proof”. In: *Mathematical Software - ICMS 2018 - 6th International Conference, South Bend, IN, USA, July 24-27, 2018, Proceedings.* Ed. by James H. Davenport et al. Vol. 10931. Lecture Notes in Computer Science. Springer, 2018, pp. 175–181. ISBN: 978-3-319-96417-1. DOI: 10.1007/978-3-319-96418-8_21. URL: https://doi.org/10.1007/978-3-319-96418-8%5C_21, William M. Farmer. *A New Style of Proof for Mathematics Organized as a Network of Axiomatic Theories.* 2018. arXiv: 1806.00810v2 [cs.LO]

6. The `PackageFormer` Prototype

From the lessons learned from spelunking in a few libraries, we concluded that metaprogramming is a reasonable road on the journey toward first-class modules in DTLs. As such, we begin by forming an 'editor extension' to Agda with an eye toward a small number of 'meta-primitives'[0] for forming combinators on modules. The extension is written in Lisp, an excellent language for rapid prototyping. The purpose of writing the editor extension is not only to show that the 'flattening' of value terms and module terms is feasible[1] ; but to also show that ubiquitous packaging combinators can be generated[2] from a small number of primitives. The resulting tool resolves many of the issues discussed in section 3.

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

[0]Section 4.3 contains an example-driven approach

[1]Indeed, the MathScheme [19] prototype already shows this.

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

This chapter is organised as follows. Firstly, the use of Lisp is explained. Then, an example demonstration of the utilities of the prototype is given. Afterwards is an overview of the combinators that we have constructed using the prototype and we showcase a few of them to solve problems observed in Chapter 3. The prototype's fundamental unit is the generalised signature of Chapter 2, with the ambient generalised type theory being MLTT (see Chapter 2); but we will not discuss how the combinators can be assigned semantics as morphisms in an appropriate category of signatures. Instead, we will reach for an Agda-based semantics in the next chapter.

Chapter Contents

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

6.1. Why an editor extension?

The prototype[3] *rewrites* Agda phrases from an extended Agda syntax to legitimate existing syntax; it is written as an Emacs editor extension to Emacs' Agda interface, using Lisp [40]. Since Agda code is predominately written in Emacs, a practical and pragmatic editor extension would need to be in Agda's de-facto IDE[5], Emacs. Moreover, Agda development involves the manipulation of Agda source code by Emacs Lisp —for example, for case splitting and term refinement tactics— and so it is natural to extend these ideas. Nonetheless, at a first glance, it is humorous[6] that a module extension for a statically dependently-typed language is written in a dynamically type checked language. However, *a lack of static types means some design decisions can be deferred as much as possible.*

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

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

Why Emacs?

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

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

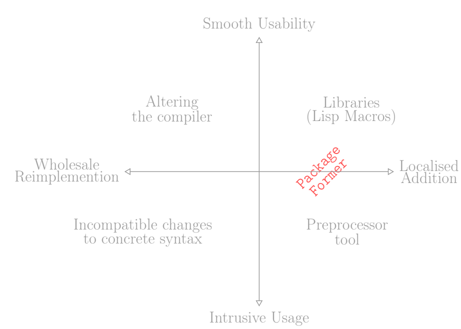
Why an editor extension? Because we quickly needed a *convenient* prototype to actually "figure out the problem".

Unless a language provides an extension mechanism, one is forced to either alter the language's compiler or to use a preprocessing tool —both have drawbacks. The former[7] is *dangerous*; e.g., altering the grammar of a language requires non-trivial propagated changes throughout its codebase, but even worse, it could lead to existing language features to suddenly break due to incompatibility with the added features. The latter is *tiresome*[8]: It can be a nuisance to remember always invoke a preprocessor before compilation or type-checking, and it becomes extra baggage to future users of the codebase —i.e., a further addition to the toolchain that requires regular maintenance in order to be kept up to date with the core language. A middle-road between the two is not always possible. However, if the language's community subscribes to *one* IDE, then a reasonable approach[9] to extending a language would be to *plug-in* the necessary preprocessing —to transform the extended language into the pure core language— in a saliently *silent* fashion such that users need not invoke it manually.

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

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

[9] "Growing a Language"; Difficulty for user setup vs difficulty for implementation



The usual workflow of an Agda user involves writing some code (types and terms alike), then asking for Agda to typecheck it. The typechecking operation is done quite frequently. Thus, one way for our prototype to fit in well to this workflow is to extend the emacs hook that triggers Agda's typechecking to also invoke our prototype.

The prototype implementation works via string manipulations. Although we have no formal proof of this, the manipulations all seem quite straightforward, and none seem to be overly time-consuming. While we can't be assured that these are linear in the size of the code, in practice, it seems like this is the case. To guard against bugs potentially introduced through this untyped "wild manipulation" phase, Agda typechecks everything that the prototype generates, thus ensuring eventual soundness.

Unlike Agda itself, which rewrites user code, such as when doing case-split, and will occasionally produce incorrect code, we eschew that. Instead, our prototype produces auxilliary files that contain Agda code, which are then imported into user code. The necessary import clauses, to the auxiliary files, are automatically inserted when not present. One benefit of this approach is that library users do not need to know about the extended language, as what is imported is pure Agda, albeit with the extended language features appearing in special comments.

Why Lisp? Emacs is extensible using Elisp[4] 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[10], to quickly see what our extended Agda syntax transpiles into.

Finally, Lisp uses a rather small number of constructs, such as macros and lambda, which themselves are used to build 'primitives', such as `defun` for defining top-level functions [48]. Knowing this about Lisp encourages us to emulate this expressive parsimony.

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

[10] E.g., since Emacs is a self-documenting editor, whenever a user of our tool wishes to see the documentation of a module combinator that they have written, or to read its Lisp elaboration, they merely need to invoke Emacs' help system —e.g., `C-h o` or `M-x describe-symbol`.

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

6.2. Aim: *Scrap the Repetition*

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

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

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

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

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

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

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

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

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

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

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

These[11] manually written ∼25 lines elaborate into the ∼100 lines of raw, legitimate, Agda syntax below —line breaks are denoted by the symbol '↪' rather than inserted manually, since all subsequent code snippets in this section are **entirely generated** by `PackageFormer`. The result is nearly a **400% increase in size**; that is, our fictitious code will save us a lot of repetition.

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

The first line declares the context of `M-Sets` using traditional Agda syntax " `record M-Set : Set₁ where` " except the we use the word `PackageFormer` to avoid confusion with the existing record concept, but[12] we also *omit* the need for a `field` keyword and *forbid* the existence of parameters. Such abstract contexts have no concrete form in Agda and so no code is generated; the second snippet above[13] shows sample declarations that result in legitimate Agda.

`PackageFormer` module combinators are called *variationals* since they provide a variation on an existing grouping mechanism. The syntax $\texttt{p} ⟴ v_1 ⟴ \cdots ⟴ v_n$ is tantamount to explicit forward func-

Now to actually use this context ...

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

⋆ ⋆ ⋆

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

⋆ ⋆ ⋆

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

⋆ ⋆ ⋆

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

⋆ ⋆ ⋆

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

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

[12] **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.

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

tion application v_n (v_{n-1} ($\cdots$ (v_1 p))). With this understanding, we can explain the different ways to organise M-sets.

In line 9, the `record` variational is invoked to transform the abstract context `M-Set` into a valid Agda record declaration, with the key word `field` inserted as necessary. Later, its first 3 fields are lifted as parameters using the meta-primitive `:waist`.

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

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

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


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


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

Notice how `Semantics𝒟` was *built from* a concrete context, namely the `Semantics` record. As such, every instance of `Semantics𝒟` can be transformed as an instance of `Semantics`: This view[14] —see Section **??**— is automatically generated and named `toSemantics` above, by default. Likewise, `Right-M-Set` was derived from `Left-M-Set` and so we have automatically have a view `Right-M-Set` → `Left-M-Set`.

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

"Arbitrary functions act on modules": When only one variational is applied to a context, the one and only sequencing operator '⟴' may be omitted. As such, the $\mathcal{D}$ecorated `Semantics𝒟` is defined as `Semantics rename f`, where `f` is the decoration function. In this form, one is tempted to believe

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

Likewise, line 13, mentions another combinator

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

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

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

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

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

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


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

Next, in line 16, we view a context as such a termtype by declaring one sort of the context to act as the termtype (carrier) and then keep only the function symbols that target it —this is the **core idea** that is used when we operate on Agda `Term`s in the next chapter.

An algebraic data type is a tagged union of symbols, terms, and so is one type —see section 5.3.

Recall from Chapter 2, symbols that target `Set` are considered sorts and if we keep only the symbols targeting a sort, we have a signature. By allowing symbols to be of type `Set`, we actually have **generalised contexts**.

Elaboration of lines 16-18 Termtypes and lawless presentations

```
{- ScalarSyntax        = M-Set ⟴ primed ⟴ data "Scalar′" -}
data ScalarSyntax : Set where
    𝟙′          : ScalarSyntax
    _×′_              : ScalarSyntax → ScalarSyntax →
    ↪  ScalarSyntax


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


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

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

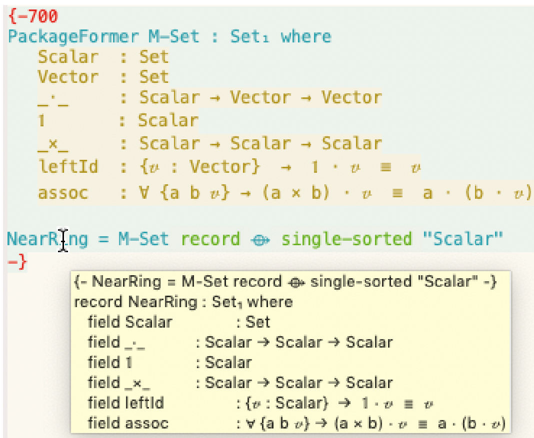
Finally, starting with line 20, declarations start with “ $\mathcal{V}$- ” to indicate that a new variation *combinator* is to be formed, rather than a new *grouping* mechanism. For instance, the user-defined `one-carrier` variational identifies both the `Scalar` and `Vector` sorts, whereas `compositional`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

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

Elaboration of lines 24-26 Conflating features gives familiar structures

```
{- LeftUnitalSemigroup = M-Set ⟴ monoidal -}
record LeftUnitalSemigroup : Set₁ where
    field Carrier                 : Set
    field _⨾_             : Carrier → Carrier → Carrier
    field 𝟙               : Carrier
    field leftId                  : {v : Carrier}  →  𝟙 ⨾ v  ≡  v
    field assoc           : {a b : Carrier} {v : Carrier} → (a ⨾ b) ⨾ v  ≡  a ⨾ (b ⨾ v)


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


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

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



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

6.3. Practicality

Herein we demonstrate how to use this system from the perspective of *library designers*. That is to say, we will demonstrate how common desirable features encountered "in the wild" —chapter 3— can be used with our system. The exposition here follows section 2 [18], reiterating many the ideas therein. These features are **not built-in** but instead are constructed from a small set of primitives, shown below, just as a small core set of language features give way to complex software programs. Moreover, users may combine the primitives — using Lisp— to **extend** the system to produce grouping mechanisms for any desired purpose.

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

Metaprogramming Meta-primitives for Making Modules

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

The few constructs demonstrated in this section not only create new grouping mechanisms from old ones, but also create morphsisms from the new, child, presentations to the old parent presentations. For example, a theory extended by new declarations comes equipped with a map that forgets the new declarations to obtain an instance of the original theory. Such morphisms are tedious to write out, and our system provides them for free. The user can implement such features using our 5 primitives —but we have implemented a few to show that the primitives are deserving of their name, as shown below.

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

Summary of Sample Variationals Provided With The System

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

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

Any variational v that takes an argument of type τ can be thought of as a **binary packaged-valued operator**,

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

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

The remainder of this section is an exposition of notable *user-defined* combinators —i.e., those which can be constructed using the system's primitives and a small amount of Lisp. Along the way, for each example, we show both the terse specfication using `PackageFormer` and its elaboration into pure typecheckable Agda. In particular, since packages are essentially a list of declarations —see Chapter 2— we begin in section 6.3.1 with the `extended-by` combinator which “grows a package”. Then, in section 6.3.2, we show

how *Agda users* can **quickly**, with a *tiny* amount of Lisp[15] knowledge, make useful variationals to abbreviate commonly occurring situations, such as a method to adjoin named operation properties to a a package. After looking at a `renaming` combinator, in section 6.3.3, and its properties that make it resonable; we show the Lisp code, in section 6.3.4 required for a pushout construction on packages. Of note is how Lisp's keyword argument feature allows the *verbose* 5-argument pushout operation to be **used** *easily* as a 2-argument operation, with other arguments optional. This construction is shown to generalise set union (disjoint and otherwise) and provide support for granular hierarchies thereby solving the so-called 'diamond problem'. Afterword, in section 6.3.5, we turn to another example of *formalising common patterns* —see Chapter 3— by showing how the idea of duality, not much used in simpler type systems, is used to mechanically produce new packages from old ones. Then, in section 6.3.6, we show how the interface segregation principle can be *applied after the fact*. Finally, we close in section 6.3.7 with a measure of the systems immediate practicality.

[15]The `PackageFormer` manual provides the expected Lisp methods one is interested in, such as `(list` x_0 `...` x_n`)` to make a list and `first, rest` to decompose it, and `(--map (···it···) xs)` to traverse it. Moreover, an Emacs Lisp cheat sheet covering the basics is provided.

6.3.1. Extension

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

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

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

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

So much repetition for an additional axiom! Eek!

As expected, the only difference is that `CommutativeMonoid`$_0$ adds a `comm`utatity axiom. Thus, given `Monoid`, it would be **more economical** to define:

Economically declaring only the new additions to 'Monoid'

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

As discussed in section 3.4, to obtain this specification of `CommutativeMonoid` in the current implementation of Agda, one would likely declare a record with two fields —one being a `Monoid` and the other being the commutativity constraint— however, this only gives the appearance of the above specification for consumers; those who produce instances of `CommutativeMonoid` are then forced to know the particular hierarchy and must provide a `Monoid` value first. It is a happy coincidence that our system alleviates such an issue; i.e., we have **flattened extensions**.

As discussed in the previous section, mouse-hovering over the left-hand-side of this declaration gives a tooltip showing the resulting elaboration, which is identical to `CommutativeMonoid`$_0$ above —followed by forgetful operation. The tooltip shows the *expanded* version of the theory, which is **what we want to specify but not what we want to enter manually**.

6.3.2. Defining a Concept Only Once

From a library-designer's perspective, our definition of `CommutativeMonoid` has the commutativity property 'hard coded' into it. If we wish to speak of commutative magmas —types with a single commutative operation— we need to hard-code the property once again. If, at a later time, we wish to move from having arguments be implicit to being explicit then we need to track down every hard-coded instance of the property then alter them —having them in-sync then becomes an issue. Instead, as shown below, the system lets us 'build upon' the `extended-by` combinator: We make an associative list of names and properties, then string-replace the meta-names *op, op', rel* with the provided user names.

The definition below uses functional methods and should not be inaccessible to Agda programmers.

⋆ ⋆ ⋆

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

⋆ ⋆ ⋆

`(pcase e (x`$_0$ `y`$_0$`) ... (x`$_n$ `y`$_n$`))` pattern matches on `e` and performs the first `y`$_i$ if `e = x`$_i$, otherwise it returns `nil`.

Writing definitions **only once** with the 'postulating' variational

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

   PROP may be a string: associative, commutative, idempotent, etc.
   Some properties require another operator or a relation; which may
   be provided via USING.

   ADJOIN-RETRACT is the optional name of the resulting retract morphism.
   Provide nil if you do not want the morphism adjoined."
   extended-by
    (s-replace "op" bop (s-replace "rel" using (s-replace "op'" using
     (pcase prop
      ("associative"   "assoc : ∀ x y z → op (op x y) z ≡ op x (op y z)")
      ("commutative"   "comm  : ∀ x y   → op x y ≡ op y x")
      ("idempotent"    "idemp : ∀ x     → op x x ≡ x")
      ("left-unit"     "unitˡ : ∀ x y z → op e x ≡ e")
      ("right-unit"    "unitʳ : ∀ x y z → op x e ≡ e")
      ("absorptive"    "absorp  : ∀ x y  → op x (op' x y) ≡ x")
      ("reflexive"     "refl    : ∀ x y  → rel x x")
      ("transitive"    "trans   : ∀ x y z → rel x y → rel y z → rel x z")
      ("antisymmetric" "antisym : ∀ x y → rel x y → rel y x → x ≡ z")
      ("congruence"    "cong    : ∀ x x' y y' → rel x x' → rel y y' → rel (op x x') (op y
      ↪  y')")
      (_ (error "𝒱-postulating does not know the property “%s”" prop))
      )))) :adjoin-retract 'adjoin-retract)
```

As such, we have a formal approach to the idea that **each piece of mathematical knowledge should be formalised only once** [39]. We can extend this database of properties as needed with relative ease. Here is an example use along with its elaboration.

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

Example Use

```
PackageFormer Magma : Set₁ where
  Carrier : Set
  _·_      : Carrier → Carrier → Carrier

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

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

Associated Elaboration

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

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

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

In conjunction with `postulating`, the `extended-by` variational makes it **tremendously easy to build fine-grained hierarchies** since at any stage in the hierarchy we have views to parent stages (unless requested otherwise) *and* the hierarchy structure is *hidden* from end-users. That is to say, ignoring the views, the above initial declaration of `CommutativeMonoid`$_0$ is identical to the `CommutativeMonoid` package obtained by using variationals, as follows.

Building fine-grained hierarchies with ease

```
PackageFormer Empty : Set1 where {- No elements -}
Type                = Empty               extended-by "Carrier : Set"
Magma               = Type                extended-by "_·_ : Carrier → Carrier → Carrier"
Semigroup           = Magma               postulating "_·_" "associative"
LeftUnitalSemigroup = Semigroup           postulating "_·_" "left-unit"  :using "▯"
Monoid              = LeftUnitalSemigroup postulating "_·_" "right-unit" :using "▯"
CommutativeMonoid   = Monoid              postulating "_·_" "commutative"
```

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

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

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

6.3.3. 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:

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

Renaming Example

```
AbealianMonoid = CommutativeMonoid renaming "_·_ to _+_"
```

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

Relationship to Parent Packages. Dual to `extended-by` which can construct (retract) views *to parent* modules mechanically, `renaming` constructs (coretract) views *from parent* packages.

That is, it has an optional argument `:adjoin-coretract` which can be provided with `t` to use a default name or provided with a string to use a desired name for the inverse part of a projection, `fromMagma` below.

Adjoining coretracts —views from parent packages

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

Sequential elaboration

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

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

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

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

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

Commutativity. Since `renaming` and `postulating` both adjoin retract morphisms, by default, we are led to wonder about the result of performing these operations in sequence 'on the fly', rather than naming each application. Since `P renaming X ⟴ postulating Y` comes with a retract `toP` via the `renaming` and another, distinctly defined, `toP` via `postulating`, we have that the operations commute if *only* the first permits the creation of a retract[16] .

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

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

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

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

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

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

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

IdempotentMagma = Magma postulating' "_⊔_" "idempotent" ⟴ renaming' "_·_ to _⊔_"
```

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

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

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

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

Simultaneous textual substitution example

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

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

Do-it-yourself. Finally, to demonstrate the accessibility of the system, we show how a generic renaming operation can be defined swiftly using the primitives mentioned listed in the first table of this section. Instead of `renaming` elements *one at a time*, suppose we want to be able to uniformly `rename` all elements in a package. That is, given a function `f` on strings, we want to map over the name component of each element in the package. This is easily done with the following declaration.

Tersely forming a new variational

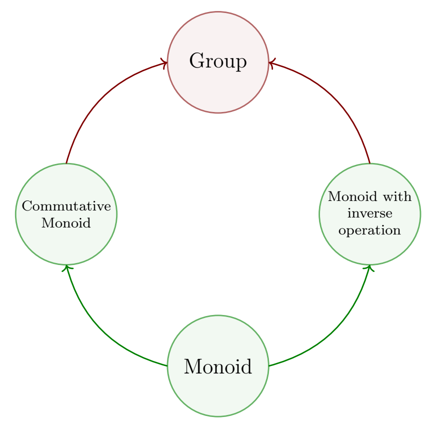
```
𝒱-rename f = map (λ element → (map-name (λ nom → (funcall f nom))) element)
```

6.3.4. Unions/Pushouts (and intersections)

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

```
CommutativeGroup₀ = Group extended-by "comm : {x y : Carrier}
↪  →  x · y  ≡  y · x" ⟴ record
```

Given green, require red



This is **problematic**: We lose the *relationship* that every commutative group is a commutative monoid. This is not an issue of erroneous hierarchical design: From `Monoid`, we could orthogonally add a commutativity property or inverse operation; `CommutativeGroup₀` then closes this diamond-loop by adding both features, as shown in the figure to the right. The simplest way to share structure is to union two presentations:

Unions of packages

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

The resulting record, `CommutativeMonoidR`, comes with three[17] 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 [18]. 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

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

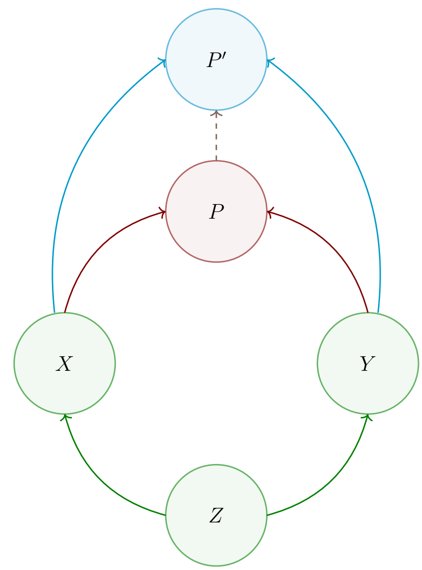
[18] Jacques Carette and Russell O'Connor. “Theory Presentation Combinators”. In: *Intelligent Computer Mathematics* (2012), pp. 202–215. DOI: 10.1007/978-3-642-31374-5_14

the `union` operation as optional keyword arguments. The more general notion of pushout is required if we were to combine[18] `Group` with `AbealianMonoid`, which have non-identical syntactic copies of `Monoid`.

[18] For example, to make rings!

What is a pushout?

The pushout of morphisms $f : Z \to X$ and $g : Z \to Y$ is, essentially, the disjoint sum of contexts X and Y where embedded elements are considered 'indistinguishable' when they share the same origin in Z via the 'paths' f and g —the pushout generalises the notion of *least upper bound* as shown in the figure to the right, by treating each '$\to$' as a '$\leq$'. Unfortunately, the resulting 'indistinguishable' elements $f(z) \approx g(z)$ are **actually distinguishable**: They may be the f-name or the g-name and a choice must be made as to which name is preferred since users actually want to refer to them later on. Hence, to be useful for library construction, the pushout construction actually requires at least another input function that provides canonical names to the supposedly 'indistinguishable' elements. Hence, 6 inputs are actually needed for forming a *usable* pushout object.



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

At first, a pushout construction needs 5 inputs, to be practical it further needs a function for canonical names for a total of 6 inputs. However, a pushout of $f : Z \to X$ and $g : Z \to Y$ is intended to be the 'smallest object P that contains a copy of X and of Y sharing the common substructure X', and as such it outputs two functions $\mathsf{inj}_1 : X \to P$, $\mathsf{inj}_2 : Y \to P$ that inject the names of X and Y into P. If we realise P as a record —a type of models— then the embedding functions are *reversed*, to obtain projections $P \to X$ and $P \to Y$: If we have a model of P, then we can forget some structure and rename via f and g to obtain models of X and Y. For the resulting construction to be useful, these names could be automated such as $\mathsf{toX} : P \to X$ and $\mathsf{toY} : P \to Y$ but such a naming scheme does not scale —but we shall use it for default names. As such, we need two more inputs to the pushout construction so the names of the resulting output functions can be used later on. *Hence, a practical choice of pushout needs 8 inputs!*

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

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

such that the resulting element has z as the chosen canonical name, then we simply require $f'\ x = z = g'\ y$.

Incidentally, using the reversed directions of f, g via f', g', we can infer the shared structure Z and the canonical name function. Likewise, by using toChild : $P \to$ Child default-naming scheme, we may omit the names of the retract functions. If we wish to rename these retracts or simply omit them altogether, we make them *optional* arguments.

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

Bimagmas: Two magmas sharing the same carrier

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

Elaboration

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

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

    field _×_     : Carrier → Carrier → Carrier

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

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

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

That is, *this particular user implementation* realises

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

as the pushout of the inclusions

`f' X₁ ∩ g' X₂ ↪ Xᵢ`

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

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

⋆ ⋆ ⋆

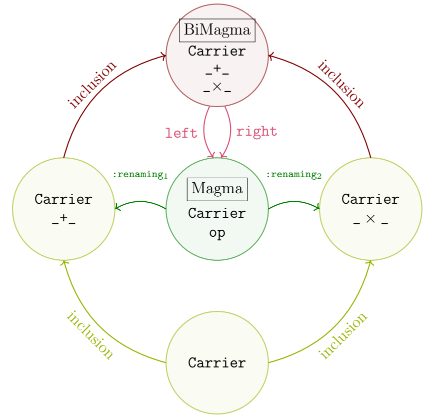
Invoke `union` with `:adjoin-retract`$_i$ `"new-function-name"` to use a new name, or `nil` instead of a string to omit the retract —as was done for `extended-by` earlier.

⋆ ⋆ ⋆

Whew, a worked-out example!

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

Given green, yield yellow, require red, form fuchsia



MagmaAgain = Magma union Magma

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

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

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

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

```
Magma'     = Magma primed ⟴ record
SumMagmas = Magma union Magma' :adjoin-retract₁ nil ⟴ record
```

Elaboration

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

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

    field Carrier' : Set
    field op'      : Carrier' → Carrier' → Carrier'

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

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

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

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

(Abridged) Pushout combinator with 4 optional arguments

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

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

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

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

     (if name-clashes
          (-let [debug-on-error nil]
            (error "%s = %s union %s \n\n\t\t → Error:
                    Elements “%s” conflict!\n\n\t\t\t%s"
                    $name $parent p (element-name (caar name-clashes))
                    (s-join "\n\t\t\t" (mapcar #'show-element (car
                    ↪  name-clashes))))))

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

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

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

1. Support for Diamond Hierarchies

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

   Three ways to get to SemiLattice

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

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

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

```
Additive           = Magma renaming "_·_ to _+_" ⟴
↪  postulating "_+_" "commutative"  :adjoin-retract nil
↪  ⟴ record

Multiplicative     = Magma renaming "_·_ to _×_"
↪  :adjoin-retract nil ⟴ record

AddMult            = Additive union Multiplicative ⟴
↪  record

AlmostNearSemiRing = AddMult ⟴ postulating "_×_"
↪  "distributiveˡ" :using "_+_" ⟴ record
```

   Elaboration

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

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

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

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

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

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

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

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

6.3.5. Duality

Maps between grouping mechanisms are sometimes called *views*, which are essentially an internalisation of the *variationals* in our system. A useful view is that of capturing the heuristic of *dual concepts*, e.g., by changing the order of arguments in an operation. Classically

The **dual**, or opposite, of a binary operation `_·_ : X → Y → Z` is the operation `_·ᵒᵖ_ : Y → X → Z` defined by `x ·ᵒᵖ y = y · x`.

in Agda, duality is *utilised* as follows:

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

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

Example

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

Continuing...

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

The RATH-Agda [53] library performs essentially this approach, for example for obtaining `UpperBounds` from `LowerBounds` in the context of an ordered set. Moreover, since category theory can serve as a foundational system of reasoning (logic) and implementation (programming), the idea of duality immediately applies to produce “two for one” theorems and programs.

The ubiquity of duality!

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

Unfortunately, this means that any record definitions in `R` must have their field names be sufficiently generic to play *both* roles of the original and the dual concept. However, well-chosen names come at an upfront cost: One must take care to provide sufficiently generic names and account for duality at the outset, irrespective of whether one *currently* cares about the dual or not; otherwise when the dual is later formalised, then the names of the original concept must be refactored throughout a library and its users. This is not the case using `PackageFormer`.

Admittedly, RATH-Agda’s names are well-chosen; e.g., `value`, `bound`$_i$, `universal` to denote a `value` that is a lower/upper `bound` of two given elements, satisfying a least upper bound or greatest lower bound `universal` property.

Consider the following heterogeneous algebra —which is essentially the main example of section 6.2 but missing the associativity field.

Left unital actions

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

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

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

Right unital actions —mechanically by duality

```
RightUnitalActionR = LeftUnitalActionR flipping "_·_" :renaming "leftId to rightId" ⟴ record
```

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

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

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

```
import Data.Product as P

PackageFormer JoinSemiLattice : Set₁ where
  Carrier : Set
  _⊑_     : Carrier → Carrier → Set

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

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

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

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

```
Checking the types of the duals

module woah (M : MeetSemiLatticeR) where
  open MeetSemiLatticeR M

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

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

6.3.6. Extracting Little Theories

The `extended-by` variational allows Agda users to easily employ the *tiny theories* [35] approach to library design: New structures are built from old ones by augmenting one concept at a time —as shown below— then one uses mixins such as `union` to obtain a complex structure. This approach lets us write a program, or proof, in a context that only provides what is *necessary* for that program-proof and nothing more. In this way, we obtain *maximal generality* for re-use! This approach can be construed as *the interface segregation*

[35] 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

principle [64, 36] : *No client should be forced to depend on methods it does not use.*

Tiny Theories Example

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

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

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

However, life is messy and sometimes one may hurriedly create a structure, then later realise that they are being forced to depend on unused methods. Rather than throw a `not implemented` exception or leave them undefined, we may use the `keeping` variational to **extract the smallest well-formed sub-PackageFormer that mentions a given list of identifiers**. For example, suppose we quickly formed `Monoid` **monolithicaly** as presented at the start of section 6.3.1, but later wished to utilise other substrata. This is easily achieved with the following declarations.

Extracting Substrata from a Monolithic Construction

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

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

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

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

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

Elaboration

```
record LeftUnitalMagma : Set₁ where

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

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

6.3.7. 200+ theories —one line for each

In order to demonstrate the **immediate practicality** of the ideas embodied by PackageFormer, we have implemented a list of mathematical concepts from universal algebra —which is useful to computer science in the setting of specifications. The list of structures is adapted from the source of a MathScheme library, which in turn was inspired by web lists of Peter Jipsen, John Halleck, and many others from Wikipedia and nLab [18, 19] . 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.

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

[18] 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

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

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

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

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

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

MathScheme's design hierarchy raised certain semantic concerns that we think are out-of-place, but we chose to leave them as is —e.g., one would think that a "partially ordered magma" would consist of a set, an order relation, and a binary operation that is monotonic in both arguments; however, PartiallyOrderedMagma instead comes with a single monotonicity axiom which is only equivalent to the two monotonicity claims in the setting of a monoidal operation.

Since the resulting **expanded code is typechecked** by Agda, we encountered a number of places where non-trivial assumptions accidentally got-by the MathScheme team. For example, in a number of places, an arbitrary binary operation occurred multiple times leading to ambiguous terms, since no associativity was declared. Even if there was an implicit associativity criterion, one would then expect multiple copies of such structures, one axiomatisation for each parenthesisation. Nonetheless, we are grateful for the source file provided by the MathScheme team.

6.4. Contributions: From Theory to Practice

The `PackageFormer` implements the ideas of Chapters 2 and 3. As such, as an editor extension, it is mostly **language agnostic** and could be altered to work with other languages such as Coq, Idris [15], and even Haskell [60]. The `PackageFormer` implementation has the following useful properties.

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

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

1. Expressive & extendable specification language for the library developer.

   - Our meta-primitives give way to the ubiquitous module combinators of Table **??**.
   - E.g., from a theory we can derive its homomorphism type, signature, its termtype, etc; we generate useful constructions inspired from universal algebra and seen in the wild —see Chapter 3.
   - An example of the freedom allotted by the extensible nature of the system is that combinators defined by library developers can, say, utilise auto-generated names when names are irrelevant, use 'clever' default names, and allow end-users to supply desirable names on demand using Lisps' keyword argument feature —see section 6.3.4.

2. Unobtrusive and a tremendously simple interface to the end user.

   - Once a library is developed using (the current implementation of) `PackageFormer`, the end user only needs to reference the resulting generated Agda, without any knowledge of the existence of `PackageFormer`.
   - We demonstrates how end-users can build upon a library by using *one line* specifications, by reducing over 1500 lines of Agda code to nearly 200 specifications using `PackageFormer` syntax.

Generated modules are necessarily 'flattened' for typechecking with Agda —see section 6.3.1.

3. Efficient: Our current implementation processes over 200 specifications in ~ 3 seconds; yielding typechecked Agda code *which* is what consumes the majority of the time.

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

4. Pragmatic: Common combinators can be defined for library developers, and be furnished with concrete syntax for use by end-users.

5. Minimal: The system is essentially invariant over the underlying type system; with the exception of the meta-primitive `:waist` which requires a dependent type theory to express 'unbundling' component fields as parameters.

6. Demonstrated expressive power *and* use-cases.

Over 200 modules are formalised as one-line specifications!

   - Common boiler-plate idioms in the standard Agda library, and other places, are provided with terse solutions using the `PackageFormer` system.
     - E.g., automatically generating homomorphism types and wholesale renaming fields using a single function —see section .

7. Immediately useable to end-users *and* library developers.

   - We have provided a large library to experiment with — thanks to the MathScheme group for providing an adaptable source file.

In the online user manual, we show how to formulate module combinators using a simple and straightforward subset of Emacs Lisp —a terse introduction to Lisp is provided.

Recall that we alluded —in the introduction to section 6.3— that we have a categorical structure consisting of `PackageFormers` as objects and those variationals that are signature morphisms. While this can be a starting point for a semantics for `PackageFormer`, we will instead pursue a *mechanised semantics*. That is, we shall encode (part of) the syntax of `PackageFormer` as Agda functions, thereby giving it not only a semantics but rather a life in a familar setting and lifting it from the status of *editor extension* to *language library*.

7. The Context Library

The PackageFormer framework is a useful tool to experiment with uncommon ways to package things together, but is relies on shuffling (untyped) strings and lacks a solid semantical basis. Instead of adding semantics after-the-fact, with the lessons learned from developing PackageFormer, we go on in this section to produce Context, an *extensible do-it-yourself packaging mechanism 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 7.3. We identify the problem and the subtleties in shifting between representations in Section 7.2.

2. Parameterised records can be obtained dynamically, on-demand, from non-parameterised records (Section 7.3) .

   - As with $\texttt{Magma}_0$, the traditional approach[20] to unbundling a record requires the use of transport along propositional equalities, with trivial reflexivity proofs —via the Σ-padding anti-pattern of Section 3.1.3. In Section 7.3, we develop a combinator, _:waist_, which removes the boilerplate necessary at the type specialisation location as well as at the instance declaration location.

3. We mechanically regain ubiquitous data structures such as ℕ, Maybe, List as the term datatypes of simple pointed and monoidal theories (Section 7.5).

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

For brevity, and accessibility, the definitions in this chapter are presented in an informal form alongside a concrete implementation *without* explanation of implementation details.

[20] Jason Gross, Adam Chlipala, and David I. Spivak. *Experience Implementing a Performant Category-Theory Library in Coq*. 2014. arXiv: 1401.7694v2 [math.CT]

A complicated Agda macro

```
accessible dashed pseudo-code
```

Code

```
... actual Agda implementation,
    requiring intimate familarity with reflection in Agda ...
```

The informal form is presented with the understanding that such functions need to be extended **homomorphically** over all possible term constructors of the host language. Enough is shown to communicate the techniques and ideas, as well as to make the resulting library usable. The details, which users do not need to bother with, are nonetheless presented so as to show how accessible these techniques are —in that, they do not require more than 15 lines per core concept.

Chapter Contents

7.1. A Tutorial on Reflection

Reflection is the ability to convert program code into an abstract syntax, a data structure that can be manipulated like any other.

Consider, for example, the tedium of writing a decidable equality for an enumerated type. Besides being tedious and error-prone, the inexpressibility of what should be a mechanically-derivable concept obscures the corresponding general principle underlying it, thus foregoing any machine assistance in ensuring any correctness or safetyness guarantees. Reflection allows a more economical and disciplined approach.

It is the aim of this section to show how to get started with reflection in Agda. To the best of my knowledge there is no up to date tutorial on this matter and, as such, we take this as an oppertunity to provide such a tutorial. Consequently, this section is reminicient of Chapter 2 on the introduction to Agda, and aims to be a self-contained presentation —occassionally demonstraing *how* various tasks may be accompalished, even though such tasks may not necessairly make an appearence in the rest of the thesis.

Necessary imports

```
module gentle-intro-to-reflection where

import Level as Level

open import Reflection hiding (name; Type)
open import Reflection.Term
open import Reflection.Pattern

open import Relation.Binary.PropositionalEquality
↪  hiding ([_])
open import Relation.Unary using (Decidable)
open import Relation.Nullary

open import Data.Unit
open import Data.Nat as Nat hiding (_⊓_)
open import Data.Bool renaming (Bool to 𝔹)
open import Data.Product
open import Data.List as List
open import Data.Char as Char
open import Data.String as String
```

There are four main types in Agda's reflection mechanism: `Name`, `Arg`, `Term`, `TC`. We will learn about them with the aid of this following simple enumerated typed, as well as other standard types.

Red, Green, Blue

```
data RGB : Set where
  Red Green Blue : RGB
```

7.1.1. `NAME` —Type of known identifiers

`Name` is the type of quoted identifiers, Agda names. Elements of this type can be formed and pattern matched using the `quote` keyword. It comes equipped with equality, ordering, and a show function. Names, along with numbers and strings, constitute the `Literal` type.

Constructing & Pattern Matching on Names

```
a-name : Name
a-name = quote ℕ

isNat : Name → 𝔹
isNat (quote ℕ) = true
isNat _         = false
```

Nope!

```
-- bad : Set → Name
-- bad s = quote s  {- s is not known -}
```

Quote will not work on function arguments; the identifier must not be a variable. This limitation is why we have a 'reflection mechanism' and not a 'macro mechanism'.

Names can be shown as strings, but are fully qualified. It would be nice to have, say, `Red` be shown as just `‘‘RGB.Red’’`. To do so, we may introduce some ‘programming’ helpers to treat Agda strings as if they where Haskell/C strings, and likewise to treat predicates as decidables. After which, we can show unqualified names by obtaining the module’s name then dropping it from the data constructor’s name.

Showing names

```
_ :   showName (quote _≡_)
    ≡ "Agda.Builtin.Equality._≡_"
_ = refl

_ :   showName (quote Red)
    ≡ "gentle-intro-to-reflection.RGB.Red"
_ = refl
```

Showing *unqualified* names

```
module-of : Name → String
module-of n = takeWhile (toDec (λ c → not (c Char.== '.')))
              ⟨𝒮⟩ showName n

_ : module-of (quote Red) ≡ "gentle-intro-to-reflection"
_ = refl

strName : Name → String
strName n = drop (1 + String.length (module-of n))
            ⟨𝒮⟩ showName n
{- The “1 +” is for the “.” separator in qualified names. -}

_ : strName (quote Red) ≡ "RGB.Red"
_ = refl
```

Programming helpers

```
{- Like “$” but for strings. -}
_⟨𝒮⟩_ : (List Char → List Char) → String →
↪  String
f ⟨𝒮⟩ s = fromList (f (toList s))

{- This should be in the standard library; I could
↪  not locate it. -}
toDec : ∀ {ℓ} {A : Set ℓ} → (p : A → 𝔹) →
↪  Decidable {ℓ} {A} (λ a → p a ≡ true)
toDec p x with p x
toDec p x | false = no λ ()
toDec p x | true = yes refl
```

Finally, if we have a name, we can obtain its fixity, which consists of its associativity —one of `assoc`^{l}, `assoc`^{r}, `non-assoc`— and its precedence —either `unrelated` or `realted` n for some ‘float’ number n. Having *fractional precedence levels* ensures that precedences are *dense*: An operator precedence can always be squeezed between any two existing precedence.

Necessary imports

```
open import Data.Float as Float using (fromℕ)

_ :   getFixity (quote _+_)
    ≡ fixity assocˡ (related (Float.fromℕ 6))
_ = refl
```

A summary of the reflection interface exposed thus far is in the table below. We use a prefix ‘⋆’ to mark elements that may be useful for programming with reflection, but are not part of Agda’s standard library for reflection. We use this star convention in the remaining sections as well.

`Name`	The type of program identifiers (excluding variables)
`quote`	Constructor for `Name`, takes an identifier as argument
`showName`	Get fully qualified string representation of a name
`_⟨𝒮⟩_`	⋆Lift a function on lists of chars to a function on strings
`toDec`	⋆Lift a Boolean into a `Decidable`
`module-of`	⋆String name of the parent module of a given `Name` argument
`strName`	⋆Unqualified string representation of a name
`getFixity`	Get the associtivity and precedence of a name

7.1.2. `Arg` —Type of arguments

Arguments in Agda may be hidden or computationally irrelevant. This information is captured by the `Arg` type.

τ-Argument ≅ Visibility × Relevance × τ

```
-- Arguments can be (visible), {hidden}, or ⦃instance⦄
data Visibility : Set where
  visible hidden instance' : Visibility

-- Arguments can be relevant or irrelevant:
data Relevance : Set where
  relevant irrelevant : Relevance

-- Arguments are characterised by their visibility & relevance
data ArgInfo : Set where
  arg-info : (v : Visibility) (r : Relevance) → ArgInfo

-- An argument of type τ is a value of τ and info about it
data Arg (τ : Set) : Set where
  arg : (i : ArgInfo) (x : τ) → Arg τ
```

Handy helpers for making argument values

```
{- visible relevant argument -}
𝓋𝓇𝒶 : {τ : Set} → τ → Arg τ
𝓋𝓇𝒶 = arg (arg-info visible relevant)

{- hidden relevant argument -}
𝒽𝓇𝒶 : {τ : Set} → τ → Arg τ
𝒽𝓇𝒶 = arg (arg-info hidden relevant)
```

Handy helpers for making variables

```
{- visible relevant variable -}
𝓋𝓇𝓋 : (debruijn : ℕ) (args : List (Arg Term))
    → Arg Term
𝓋𝓇𝓋 n args = 𝓋𝓇𝒶 (var n args)

{- hidden relevant variable -}
𝒽𝓇𝓋 : (debruijn : ℕ) (args : List (Arg Term))
    → Arg Term
𝒽𝓇𝓋 n args = 𝒽𝓇𝒶 (var n args)
```

So much for reflected arguments.

In the next section we will turn to variables —which live in the `Term` datatype. Variables are arguments —i.e., entities with a visibility and relevance— whose payload is a natural number (along with a list of arguments); this *nameless variables* approach is known as *De Bruijn indexing*. The index n refers to the argument that is n locations away from 'here'.

Given a 'usual' λ-term `t`, its De Bruijn index presentation is $\emptyset\ /_0$ `t` where the $\Gamma\ /_n$ `s` has Γ denoting "the bound variables encountered thus far" and n denotes "the depth, how many lambdas have been encountered". For example,

$$\emptyset\ /_0\ (\lambda\ f.\ \lambda\ g.\ \lambda\ x.\ f\ x\ (g\ x)) \quad = \quad \lambda\ \lambda\ \lambda\ 2\ 0\ (1\ 0)$$

Notice that the first '2' refers to the variable bound by the λ that is "2 lambdas away".

Mechanically going nameless

```
-- The τᵢ are existing λ-terms
Usual-λ-Term ::= x | τ₁ τ₂ | (λ x • τ₃)

-- Treating contexts Γ as functions, as in Ch2,
-- with comma for function extension (patching)

-- For variables x
Γ /ₙ x  =  if x ∈ domain Γ then n - Γ(x) else x fi

-- For abstractions
Γ /ₙ (λ x • e)  =  λ  (Γ, (x, n)) /ₙ₊₁ e

-- For applications
Γ /ₙ (s t)      =  (Γ /ₙ s) (Γ /ₙ t)
```

`Arg` τ	A value of type τ along with its visibility and relevance Example: `arg (arg-info visibile relevant) 3`
vra `e`	⋆Constructs a *v*isibile *r*elevant *a*rgument with value `e`
hra `e`	⋆Constructs a *h*idden *r*elevant *a*rgument with value `e`
vrv `n args`	⋆Constructs a *v*isible *r*elevant *v*ariable with debruijn index `n` and arguments `args`
hrv `n args`	⋆Constructs a *h*idden *r*elevant *v*ariable with debruijn index `n` and arguments `args`

7.1.3. Term —Type of terms

The quoteTerm keyword is used to turn a well-typed fragment of code —concrete syntax— into a value of the Term datatype —abstract syntax tree (AST). Before any examples, here is the definition of Term.

Abstract Syntax Trees —Reflected Terms

```
data Term where

  var      : (x : ℕ)  (args : List (Arg Term)) → Term

  con      : (c : Name) (args : List (Arg Term)) → Term
  def      : (f : Name) (args : List (Arg Term)) → Term

  lam      : (v : Visibility) (t : Abs Term) → Term
  pat-lam  : List Clause → List (Arg Term) → Term

  -- Telescopes, or function types; λ-abstraction for types.
  pi       : (a : Arg Type) (b : Abs Type) → Term

  -- "Set n" or some term that denotes a type
  agda-sort : (s : Sort) → Term

  -- Metavariables; introduced via quoteTerm
  meta     : (x : Meta) → List (Arg Term) → Term

  -- Literal  ≅  ℕ | Word64 | Float | Char | String | Name |
  ↪  Meta
  lit      : (l : Literal) → Term

  -- Items not representable by this AST; e.g., a hole.
  unknown  : Term {- Treated as '_' when unquoting. -}
```

A variable has a De Bruijn index and may be applied to arguments.

Constructors and definitions may be applied to a list of arguments.

λ-abstractions bind one variable, t is the variable name along with the λ-body.

⋆ ⋆ ⋆

Abs A ≅ String × A

Sort ≅ LevelTerm | ℕ | unknown

Clause ≅ List (Arg Pattern) × Term
       | List (Arg Pattern

Pattern ≅ "con Name (List (Arg Pattern))"
        | Literal |"proj Name"
        | "absurd" | "var String"

An example reflected term is in the following snippet. Even though the *concrete syntax* for propositional equalities takes two *v*isible *r*elevant *a*rguments —the left side and right side—, the resulting *abstract syntax* tree exposes the fact that there are actually an *additional* two *h*idden *r*elevant *a*rguments that happen to be inferred: The common type of the explicit arguments and the level of said type. The propositional equality is a defined name; whose *h*idden arguments also happen to be defined names, whereas its *v*isibile arguments are literal strings.

The reflected term could be presented more compactly by invoking quoteTerm in the AST.

Reflecting a partially-applied type

```
_ :  quoteTerm _≡_
   ≡ def (quote _≡_) []
_ = refl

_ : quoteTerm (_≡_ "1")
  ≡ def (quote _≡_) ( hra (quoteTerm Level.zero)
                    :: hra (quoteTerm String)
                    :: vra  (quoteTerm "1")
                    :: [])
_ = refl
```

Reflecting a fully-applied type

```
_ : quoteTerm ("l" ≡ "r") ≡ def (quote _≡_)
                              (  hra (def (quote Level.zero) [])
                              :: hra (def (quote String)     [])
                              :: vra (lit (string "l"))
                              :: vra (lit (string "r"))
                              :: [])
_ = refl
```

The above is not the *section* "l" ≡_ ! Sections are syntactic abbreviations for λ-abstractions! Keep reading ;-)

Besides defined names and literals, we may also reflect constructors and use polymorphism; as shown below.

Constructors and Polymorphism

```
_ : quoteTerm 1 ≡ lit (nat 1)
_ = refl

_ :    quoteTerm (suc zero)
    ≡ con (quote suc) (vra (quoteTerm zero) :: [])
_ = refl

_ : quoteTerm true ≡ con (quote true) []
_ = refl


_ : ∀ {level : Level.Level}{Type : Set level} (x y : Type)
   →   quoteTerm (x ≡ y)
     ≡ def (quote _≡_)
         (hrv 3 [] :: hrv 2 [] :: vrv 1 [] :: vrv 0 [] :: [])

_ = λ x y → refl
```

A *constructor*, well, constructs a value of an algebraic data type; whereas a *defined name* is a (possibly nullary) user-defined function (including type formers). Unlike functions, constructors have no computation, reduction, rules.

As discussed in the previous section, a De Bruijn index n refers to the lambda variable that is "n lambdas away" from its use site. For example, *vrv* 1 means starting at the position where *vrv 1* occurs in the text, go 1 lambdas away thereby getting the variable `x`: The first lambda away is `(y : Type)` and so the second lambda away is `(x : Type)`. (Scoped declarations are an abbreviation for multiple declarations, as discussed in Chapter 2.)

With the above example mentioning variables, it is natural to consider representing λ-abstractions as `Term` values. For example, a simple identity function, say, on the Booleans ($\lambda x : \mathbb{B} \bullet x$) consists of a lambda with a *visible abstract* argument named "x" along with a body merely being the 0-nearest bound variable, applied to an empty list of arguments. Below is a slightly more complex example.

Reflecting a λ

```
_ :   quoteTerm (λ (x : 𝔹) → x)
    ≡ lam visible (abs "x" (var 0 []))
_ = refl
```

Eek! Reflected λs are untyped! We'll return to this later!

Reflecting a function application operator —brutally

```
_ : quoteTerm (λ (a : ℕ) (f : ℕ → ℕ) → f a)
   ≡  lam visible (abs "a"
        (lam visible (abs "f"
          (var 0 (arg (vra (var 1 []) :: [])))))
_ = refl
```

The application, `f a`, is represented as the variable 0 lambdas away from the body applied to the variable 1 lambdas away from the body.

This is rather messy, but it can be made more readable by the aid of some syntactic sugar.

Reflecting a function application operator —elegantly

```
_ :   quoteTerm (λ (a : ℕ) (f : ℕ → ℕ) → f a)
    ≡ λv "a" ↦ λv "f" ↦ var 0 [ 𝑣𝑟𝑎 (var 1 []) ]
_ = refl
```

λs with *v*isibile and *h*idden arguments

```
infixr 5 λv_↦_  λh_↦_

λv_↦_  λh_↦_ : String → Term → Term
λv x ↦ body  = lam visible (abs x body)
λh x ↦ body  = lam hidden (abs x body)
```

Much easier on the eyes, hands, and brains!

Using these syntactic abbreviation, we can quickly compare how λ-arguments can be "shunted" into a quotation, as follows for the constant function.

Delicious, delicious, (syntactic) sugar!

Shunting the "waist" of a constant function

```
_ : {A B : Set} →   quoteTerm (λ (a : A) (b : B) → a)
                  ≡ λv "a" ↦ (λv "b" ↦ var 1 [])
_ = refl

_ :   quoteTerm (λ {A B : Set} (a : A) (_ : B) → a)
    ≡ λh "A" ↦ λh "B" ↦ λv "a" ↦ λv "_" ↦ var 1 []
_ = refl
```

λ-terms are governed by the rules below. Such terms are formed by the *λ-abstraction rule*: If $E : \beta$ whenever $x : \alpha$, then $(\lambda x \to E) : (\alpha \to \beta)$. Their 'computation' is captured by the β-rule and 'δefinition lookup' is captured by the δ-rule.

η-rule: $(\lambda x \to f\, x) = f$

β-rule: $(\lambda x \to E)\, v = E[x \coloneqq v]$

δ-rule: $f\, v = E[x \coloneqq v]$ for $f = (\lambda x \to E)$

We can now return to the above remark about reflecting *sections*: For a binary operation `_⊕_` $: \alpha \to \beta \to \gamma$, its *left section* by any value `a` $: \alpha$ is the function `(λ b → a ⊕ b)` $: \beta \to \gamma$, which is generally denoted by `a ⊕_` or, informally by $(a\oplus)$. Likewise for right sections.

Left Sections: No λv after normalisation

```
_ :   quoteTerm ("l" ≡_)
    ≡ def (quote _≡_)
          ( hra (quoteTerm Level.zero)
          :: hra (quoteTerm String)
          :: vra (quoteTerm "l")
          :: [])
_ = refl
```

Right Sections: Required λv

```
_ :   quoteTerm (_≡ "r")
    ≡ λv "section" ↦
       def (quote _≡_)
           ( hra (quoteTerm Level.zero)
           :: hra (quoteTerm String)
           :: vra (var 0 [])
           :: vra (quoteTerm "r")
           :: [])
_ = refl
```

As the above example shows, quotation automatically performs η-reduction. The relationships of `quoteTerm` with λ's governing rules are summarised as follows —including the above 'argument-shunting' observation.

Helper for concrete examples below

```
id : {A : Set} → A → A
id x = x
```

Shunting Law —"`quoteTerm` computation rule"

`quoteTerm (λ (x : τ) → e)   ≡   λv "x" ↦ quoteTerm e`

Eta Law

`quoteTerm (λ x → f x)   ≡   quoteTerm f`

η in action

```
_ :   quoteTerm (λ (x : ℕ) → id x)
    ≡ def (quote id) (hra (quoteTerm ℕ) :: [])
_ = refl
```

Beta Law

`quoteTerm` typechecks and normalises its argument before yielding a `Term` value.

β in action!

```
_ :   quoteTerm ((λ x → x) "nice")
    ≡  lit (string "nice")
_ = refl
```

No Delta Law

quoteTerm does no δ-reduction: Function definitions are not elaborated.

δ *not* in action!

```
_ :    quoteTerm (id "a")
   ≡  def (quote id)
          ( hra (quoteTerm String)
          :: vra (quoteTerm "a")
          :: [])
_ = refl
```

A relationship between quote and quoteTerm!

Since δ-reduction does not happen, known names f in a quoted term are denoted by a quote f —since no δefinitional elaboration happens— in the AST representation; as shown below.

No δ-reduction for top-level defined names

```
f : ℕ → ℕ
f x = x

_ : quoteTerm f ≡ def (quote f) []
_ = refl
```

In contrast, names that *vary* are denoted by a var term constructor in the AST representation.

Names that *vary* are reflected as var terms

```
module _ {A B : Set} {f : A → B} where

  _ : quoteTerm f ≡ var 0 []
  _ = refl
```

Local names are *not* considered top-level defined names.

lets give rise to vars

```
_ : let f₁ : ℕ → ℕ; f₁ x = x
    in  quoteTerm f₁  ≡  λv "x" ↦ var 0 []
_ = refl
```

As such, we could form a module and let rules for quoteTerm —e.g., the latter could be let x = E in quoteTerm P = quoteTerm (P[x ≔ E]).

quoteTerm	Reify concrete Agda syntax as Term values, ASTs
$\lambda v\_{\mapsto}\_$ and $\lambda h\_{\mapsto}\_$	⋆Make lam-da Term values with *v*isibile, or *h*idden, arguments

7.1.4. Metaprogramming with the Type-Checking Monad TC

A monadic interface to Agda's 'T'ype'C'hecking utility is available through the TC type former. Below are a few notable (postulated) bindings to the typechecking utility; the offical Agda documentation pages mention further primitives for the current context, type errors, and metavariables.

Since `TC : ∀ {ℓ} → Set ℓ → Set ℓ` is a monad, we may use `do`-notation when forming typechecking computations.

Interface to Agda's Typechecker

```
{- Take what you have and try to make it fit
   into the current goal. -}
unify : (have : Term) (goal : Term) → TC ⊤

{- Try first computation;
   if it crashes with a type error, try the second. -}
catchTC : ∀ {a} {A : Set a} → TC A → TC A → TC A

{- Infer the type of a given term. -}
inferType : Term → TC Type

{- Check a term against a given type. -}
checkType : Term → Type → TC Term

{- Compute the normal form of a term. -}
normalise : Term → TC Term

{- Quote a value, returning the corresponding Term. -}
quoteTC : ∀ {a} {A : Set a} → A → TC Term

{- Unquote a Term, returning the corresponding value. -}
unquoteTC : ∀ {a} {A : Set a} → Term → TC A

{- Declare a new function of the given type.  -}
declareDef : Arg Name → Type → TC ⊤

{- Define a declared function. -}
defineFun : Name → List Clause → TC ⊤

{- Get the type of a defined name. -}
getType : Name → TC Type

{- Get the definition of a defined name. -}
getDefinition : Name → TC Definition
```

Warning: There's a `freshName : String → TC Name` primitive, which is, currently, *mostly* useless: It seems that the scope checker runs before any reflection code and so any names exposed by reflection code are "not in scope" when the scope checker runs. Since scope checking is a crucial component of type checking, a possible workaround would be to have multiple phases of scope and type checking with message passing occurring between the checkers.

`checkType` checks a term against a given type. This may resolve implicit arguments in the term, so a new refined term is returned.

For `declareDef`, the function must be defined later using `defineFun`. For `defineFun`, the function may have been declared using `declareDef` or with an explicit top-level type signature.

TC computations, or *metaprograms*, can be run by declaring them as *macros* or by unquoting. Let us begin with the former.

7.1.5. Unquoting —Making new functions & types

Recall our RGB example type was a simple enumeration consisting of Red, Green, Blue. Consider the singleton type, predicate, IsRed whose only inhabitant is Red. The name Red completely determines this datatype; so let's try to generate it mechanically. Unfortunately, as far as I could tell, there is currently no way to unquote data declarations. As such, we'll settle for its isomorphic functional formulation. Below, the unquoteDecl keyword allows us to obtain a Name value, say IsRed. We then quote the desired type, τ, declare a function of that type, then define it using the provided Name.

Using Agda's syntactic sugar

```
data IsRed : RGB → Set where
  yes : IsRed Red
```

No sugar

```
IsRed : RGB → Set
IsRed x = x ≡ Red
```

Unquoting a singelton type predicate

```
unquoteDecl IsRed =
  do τ ← quoteTC (RGB → Set)
     declareDef (𝑣𝑟𝑎 IsRed) τ
     defineFun IsRed
        [ clause [ 𝑣𝑟𝑎 (var "x") ]
                 (def (quote _≡_)
                      (‘ℓ₀ :: ‘RGB :: ‘Red :: 𝑣𝑟𝑣 0 [] :: []))]
```

For readability, let's quote the relevant parts.

Quoted abbreviations

```
‘ℓ₀ : Arg Term
‘ℓ₀ = 𝒉𝒓𝒂 (def (quote Level.zero) [])

‘RGB : Arg Term
‘RGB = 𝒉𝒓𝒂 (def (quote RGB) [])

‘Red : Arg Term
‘Red = 𝑣𝑟𝑎 (con (quote Red) [])
```

Let's try out our newly *unquote* declared type!

```
red-is-a-solution : IsRed Red
red-is-a-solution = refl

green-is-not-a-solution : ¬ (IsRed Green)
green-is-not-a-solution = λ ()

red-is-only-solution : ∀ {c} → IsRed c → c ≡ Red
red-is-only-solution refl = refl
```

There is a major problem with using unquoteDef outright like this: We cannot step-wise refine our program using holes *{! !}*, since that would result in unsolved meta-variables. Instead, we split this process into two stages: A programming stage, then an unquotation stage.

A generalised 2-stage process to unquotation

```
-- ⟨0⟩ Definition stage, we can use ‘?’ as we form this program
define-Is : Name → Name → TC ⊤
define-Is is-name qcolour
    = defineFun is-name
        [ clause [ 𝑣𝑟𝑎 (var "x") ]
                 (def (quote _≡_)
                      (‘ℓ₀ :: ‘RGB :: 𝑣𝑟𝑎 (con qcolour []) :: 𝑣𝑟𝑣
                      ↪  0 [] :: []))]

-- ⟨1⟩ Unquotation stage with a *mandatory* type declaration
IsRed' : RGB → Set
unquoteDef IsRed' = define-Is IsRed' (quote Red)

-- ⟨2⟩ Usage state: Trying it out
_ : IsRed' Red
_ = refl
```

Notice that if we use unquoteDef, we must provide a type signature. We only do so for illustration; the next code block avoids such a redundancy by using unquoteDecl. The above general approach lends itself nicely to the other data constructors as well:

Unquoting multiple singelton predicate types

```
-- ⟨0⟩′ Definition stage *with* a type declaration.
declare-Is : Name → Name → TC ⊤
declare-Is is-name qcolour =
  do let η = is-name
     τ ← quoteTC (RGB → Set)
     declareDef (𝑣𝑟𝑎 η) τ
     define-Is is-name qcolour
     defineFun is-name
        [ clause [ 𝑣𝑟𝑎 (var "x") ]
                 (def (quote _≡_) (‘ℓ₀ :: ‘RGB :: 𝑣𝑟𝑎 (con
                 ↪  qcolour []) :: 𝑣𝑟𝑣 0 [] :: []))]

-- ⟨1⟩′ Unquotation stage, in one line.
unquoteDecl IsBlue  = declare-Is IsBlue  (quote Blue)
unquoteDecl IsGreen = declare-Is IsGreen (quote Green)

{- Example use -}
disjoint-rgb  : ∀{c} → ¬ (IsBlue c × IsGreen c)
disjoint-rgb (refl , ())
```

The next natural step is to avoid manually invoking `declare-Is` for each constructor. Unfortunately, as disucussed earlier, fresh names are not accessible, since they come into scope *after* typechecking.

7.1.6. Example: Avoid tedious `refl` proofs

We are now in a position to tackle a ‘real-world’ situation.

When functions perform a lot of pattern matching, then to prove properties about them, it becomes necessary to pattern match on the arguments they pattern match against —so that a particular clause of the function applies. For instance, consider the following two functions with overly excessive pattern matching.

Too much pattern matching...

```
just-Red : RGB → RGB
just-Red Red   = Red
just-Red Green = Red
just-Red Blue  = Red

only-Blue : RGB → RGB
only-Blue Blue = Blue
only-Blue _   = Blue
```

Then, to show that the above function `just-Red` is constantly `Red` requires pattern matching then a `refl` *for each clause*. Likewise, for `just-Blue`.

...results in more pattern matching

```
just-Red-is-constant : ∀{c} → just-Red c ≡ Red
just-Red-is-constant {Red}   = refl
just-Red-is-constant {Green} = refl
just-Red-is-constant {Blue}  = refl

{- Yuck, another tedious proof -}
only-Blue-is-constant : ∀{c} → only-Blue c ≡ Blue
only-Blue-is-constant {Blue}  = refl
only-Blue-is-constant {Red}   = refl
only-Blue-is-constant {Green} = refl
```

In such cases, we can encode the general design decisions —*pattern match and yield refl*— then apply the schema to each use case. Here is the schema:[21]

Factoring out the insight

```
constructors : Definition → List Name
constructors (data-type pars cs) = cs
constructors _ = []

by-refls-on : Name → Name → Term → TC ⊤
by-refls-on δατατγρε nom thm-you-hope-is-provable-by-refls
 = let mk-cls : Name → Clause
       mk-cls qcolour = clause [ hra (con qcolour []) ]
                               ( con (quote refl) []   )
   in
   do let η = nom
      δ ← getDefinition δατατγρε
      let clauses = List.map mk-cls (constructors δ)
      declareDef (vra η) thm-you-hope-is-provable-by-refls
      defineFun η clauses
```

[21] Now, `unquoteDecl f = by-refls-on τ f (quote P)` results in the following function —where the c_i are the constructors of τ.

Elaboration of 'by-refls-on'

```
f : ∀ {e : τ} → P
 ↪  e
f c₁  = refl
 ⋮    ⋮  ⋮
f cₙ  = refl
```

Here is a use case.

Factoring out the insight

```
obviously : Name → Term → TC ⊤
obviously = by-refls-on (quote RGB)

_ : ∀{c} → just-Red c ≡ Red
_ = nice
  where unquoteDecl nice = obviously nice (quoteTerm (∀{c} →
  ↪  just-Red c ≡ Red))
```

Where,

1. The first `nice` refers to the function created by the right-hand side (RHS) of the unquote.

2. The RHS `nice` refers to the `Name` value provided by the left-

hand side (LHS).

3. The LHS `nice` is a declaration of a `Name` value.

This is rather clunky since the theorem to be proven was repeated twice —repetition is a signal that something's wrong! In the next section we use macros to avoid such repetiton, as well as the `quoteTerm` keyword.

Warning! We use a `where` clause since unquotation cannot occur in a `let`.

Here's another use case of the proof pattern

Factoring out the insight

```
_ : ∀{c} → only-Blue c ≡ Blue
_ = nice
  where unquoteDecl nice = obviously nice (quoteTerm ∀{c} →
  ↪  only-Blue c ≡ Blue)
```

One proof pattern, multiple invocations!

7.1.7. Macros —Abstracting Proof Patterns

Macros are functions of type $\tau_0 \to \tau_1 \to \cdots \to$ `Term` $\to$ `TC` $\top$ that are defined in a `macro` block. The last argument is supplied by the type checker and denotes the "goal" of where the macro is placed: One generally unifies what they have with the goal, what is desired in the use site. In contrast to splicing terms with `unquoteDecl`, Agda *macros* have the following benefits:

1. Metaprograms can be run in a term position.

2. Without the macro block, we run computations using the `unquote` and `unquoteDecl` keyphrases.

3. Quotations are performed automatically; e.g., if `f : Term` $\to$ `Name` $\to$ $\mathbb{B}$ $\to$ `Term` $\to$ `TC` $\top$ then an application `f u v w` desugars into `unquote (f (quoteTerm u) (quote v) w)`.

4. No syntactic overhead: Macros are applied like normal functions.

Macros cannot be recursive; instead one defines a recursive function outside the macro block then has the macro call the recursive function.

1. C-style macros In the C language one defines a macro, say, by `#define luckyNum 1729` then later uses it simply by the name `luckyNum`. Without macros, we have syntactic overhead using the `unquote` keyword:

Factoring out the insight

```
luckyNum₀ : Term → TC ⊤
luckyNum₀ goal = unify goal (quoteTerm 1729)

num₀ : ℕ
num₀ = unquote luckyNum₀
```

Instead, we can achieve C-style behaviour by placing our metaprogramming code within a `macro` block.

Factoring out the insight

```
macro
  luckyNum : Term → TC ⊤
  luckyNum goal = unify goal (quoteTerm 1729)
num = luckyNum
```

Unlike C, all code fragments must be well-defined.

2. Tedious Repetitive Proofs No More! Suppose we wish to prove that addition, multiplication, and exponentiation have right units 0, 1, and 1 respectively. We obtain the following nearly identical proofs.

Factoring out the insight

```
+-rid : ∀{n} → n + 0 ≡ n
+-rid {zero}  = refl
+-rid {suc n} = cong suc +-rid

*-rid : ∀{n} → n * 1 ≡ n
*-rid {zero}  = refl
*-rid {suc n} = cong suc *-rid

^-rid : ∀{n} → n ^ 1 ≡ n
^-rid {zero}  = refl
^-rid {suc n} = cong suc ^-rid
```

There is clearly a pattern here screaming to be abstracted, let's comply. The natural course of action in a functional language is to try a higher-order combinator:

Factoring out the insight

```
{- “for loops” or “Induction for ℕ” -}
foldn : (P : ℕ → Set) (base : P zero) (ind : ∀ n → P n
↪   → P (suc n))
      → ∀(n : ℕ) → P n
foldn P base ind zero    = base
foldn P base ind (suc n) = ind n (foldn P base ind n)
```

Now the proofs are shorter:

Factoring out the insight

```
_ : ∀ (x : ℕ) → x + 0 ≡ x
_ = foldn _ refl (λ _ → cong suc)    {- This and next
↪  two are the same -}

_ : ∀ (x : ℕ) → x * 1 ≡ x
_ = foldn _ refl (λ _ → cong suc)    {- Yup, same proof
↪  as previous -}

_ : ∀ (x : ℕ) → x ^ 1 ≡ x
_ = foldn _ refl (λ _ → cong suc)    {- No change, same
↪  proof as previous -}
```

Unfortunately, we are manually copy-pasting the same proof *pattern*.

> When you see repetition, copy-pasting, know that there is room for improvement!
>
> Don't repeat yourself!

Repetition can be mitigated a number of ways, including typeclasses or metaprogramming, for example. The latter requires possibly less thought and it's the topic of this article, so let's do that. Rather than use unquotes and their syntactic overhead, we use macros instead. The definition below essentially produce the repeated proofs, `foldn P refl (λ _ → cong suc)`, at each call.

Factoring out the insight

```
macro
  _trivially-has-rid_ : (let A = ℕ) (_⊕_ : A → A → A)
  ↪  (e : A) → Term → TC ⊤
  _trivially-has-rid_ _⊕_ e goal
   = do τ ← quoteTC (λ(x : ℕ) → x ⊕ e ≡ x)
        unify goal (def (quote foldn)             {- Using
        ↪  foldn    -}
          ( vra τ                                 {- Type
          ↪  P          -}
          :: vra (con (quote refl) [])            {- Base
          ↪  case       -}
          :: vra (λv "_" ↦ quoteTerm (cong suc))  {-
          ↪  Inductive step -}
          :: []))
```

Now the proofs have minimal repetition *and* the proof pattern is written only *once*:

Factoring out the insight

```
_ : ∀ (x : ℕ) → x + 0 ≡ x
_ = _+_ trivially-has-rid 0

_ : ∀ (x : ℕ) → x * 1 ≡ x
_ = _*_ trivially-has-rid 1

_ : ∀ (x : ℕ) → x * 1 ≡ x
_ = _^_ trivially-has-rid 1
```

7.2. The Problems

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

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

Dynamical Systems

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

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

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

Each $\texttt{DynamicSystem}_i$ is a type constructor of `i`-many arguments; but it is **the types of these constructors that provide insight into the sort of data they contain** as shown in the following table and discussed in Sections 3.1.3 and 3.1.

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

Recall, say from Section 4.1, that we refer to the concern of moving from a record to a parameterised record as **the unbundling problem**[22]. For example, moving from the *type* $\texttt{Set}_1$ to the *function type* Π X : Set • Set gets us from $\texttt{DynamicSystem}_0$ to something resembling $\texttt{DynamicSystem}_1$, which we arrive at if we can obtain a *type constructor* of the form λ X : Set • ⋯. We shall refer to the latter change as *reïfication* since the result is more con-

[22]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

crete: It can be applied. This transformation will be denoted by Π→λ. To clarify this subtlety, consider the following forms of the *type* of the polymorphic identity function. Notice that $\mathtt{id}\tau_i$ *exposes* i-many details at the type level to indicate the sort of data it consists of. However, notice that $\mathtt{id}_0$ is a **type of functions** whereas $\mathtt{id}_1$ is a **function on types**. Indeed, the final form is derived from the first one: $\mathtt{id}\tau_2$ = Π→λ $\mathtt{id}\tau_0$. This equation is true by `refl`exivity, as shown below.

Polymorphic Identity Functions

```
idτ₀ : Set₁
idτ₀ = Π X : Set • Π e : X • X

idτ₁ : Π X : Set • Set
idτ₁ = λ (X : Set) → Π e : X • X

idτ₂ : Π X : Set • Π e : X • Set
idτ₂ = λ (X : Set) (e : X) → X

{- Surprisingly, the latter is derivable from the former -}
_ : idτ₂ ≡ Π→λ idτ₀
_ = refl

{- The relationship with idτ₁ is clarified later when we get to _:waist_ -}
```

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.

7.3. Monadic Notation

There is little use in an idea that is difficult to use in practice. As such, we conflate records and termtypes by starting with an ideal syntax they would share, then derive the necessary artefacts that permit it. As discussed at the start of the chapter, our choice of syntax is monadic `do`-notation [68, 63]:

Idealised syntax for one source of truth

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

Here `Context`, `End`, and the underlying monadic bind operator are unknown. Since we want to be able to *expose* a number of fields at will, we may take `Context` to be types indexed by a number denoting exposure. Moreover, since records are product types, we expect there to be a recursive definition whose base case will be the identity of products, the unit type 𝟙 —which corresponds to ⊤ in the Agda standard library and to `()` in Haskell. The following table shows example exposure 'waists' for the `DynamicSystem` context.

Elaborations of `DynamicSystem` at various exposure levels

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

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

Context and End

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

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

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

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

Defining Bind —First Attempt

```
-- >>= : ∀ {A B : Set}   →        m A         →           (A    → m B)        → m B
_>>=_   : ∀ {a b : Level} → (Γ : Context a) → (∀ {n} → Γ n → Context b) → Context (a
↪  ⊎ b)
(Γ >>= f) zero     = Σ γ : Γ 0 • f γ 0
(Γ >>= f) (suc n)  = Π γ : Γ n • f γ n
```

The definition here accounts for the current exposure index: If zero, we have *record types*, otherwise *function types*. Using this definition, the above dynamical system context would need to be expressed using the lifting quote operation.

The extensibility of `Context` is provided by the definition of bind: Rather than Σ and Π, users may use or augment the framework in other forms —e.g., Π^w, $\mathcal{W}$, or let⋯in⋯ (as shown in $\mathcal{N}_1$' below) *or combinations thereof.

Example Use

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

{- or -}

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

Interestingly[23,24], use of `do`-notation in preference to bind, >>=, was suggested by John Launchbury in 1993 and was first implemented by Mark Jones in Gofer. Anyhow, with our goal of practicality in mind, we shall "build the lifting quote into the definition" of bind:

The Definition of Bind

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

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

[23] Richard Bird. "Thinking Functionally with Haskell". In: (2009). DOI: 10.1017/cbo9781316092415. URL: http://dx.doi.org/10.1017/cbo9781316092415

[24] 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

Factories and Instantiation —Natural numbers form a dynamic system

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

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

{- ‘Instantiaing’ State to be ℕ in “DynamicSystem 1” -}

𝒩₁′ : let State = ℕ in Σ start : State  • Σ s : (State → State)  • 𝟙 {ℓ₀}
𝒩₁′ = 0 , suc , tt
```

To get from $\mathcal{N}_1$ to $\mathcal{N}_1{}'$, it seems what we need is a method, say Π→λ, that takes a Π-type and transforms it into a λ-expression. One could use a universe, an algebraic type of codes denoting types, to define Π→λ. However, one can no longer then easily use existing types since they are not formed from the universe's constructors, thereby resulting in duplication of existing types via the universe encoding. This is neither practical nor pragmatic. As such, we are left with pattern matching on the language's type formation primitives as the only reasonable approach. The method Π→λ is thus a macro[25] that acts on the syntactic term representations of types. Below is the main transformation.

Π→λ

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

Source

```
Π→λ-type : Term → Term
Π→λ-type (pi a (abs x b)) = pi a  (abs x (Π→λ-type b))
Π→λ-type x = unknown

Π→λ-helper : Term → Term
Π→λ-helper (pi a (abs x b)) = lam visible (abs x (Π→λ-helper b))
Π→λ-helper x = x

macro
  Π→λ : Term → Term → TC Unit.⊤
  Π→λ tm goal =  normalise tm
                >>=term λ tm' → checkType goal (Π→λ-type tm')
                >>=term λ _ →  unify goal (Π→λ-helper tm')
```

That is, we walk along the term tree replacing (consecutive) occurrences of Π with λ; as shown

[25] 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.

in the following *formal* (i.e., typechecked) calculation.

Example use of Π→λ

```
_ = Π→λ (DynamicSystem 2)
  ≡⟨ "Definition of DynamicSystem at exposure level 2" ⟩'
   Π→λ (Π X : Set • Π s : X • Σ n : (X → X)  • 𝟙 {ℓ₀})
  ≡⟨ "Definition of Π→λ; replace a ‘Π’ by a ‘λ’" ⟩'
   (λ (X : Set) → Π→λ (Π s : X • Σ n : (X → X)  • 𝟙 {ℓ₀}))
  ≡⟨ "Definition of Π→λ; replace a ‘Π’ by a ‘λ’" ⟩'
   (λ (X : Set) → λ (s : X) → Π→λ (Σ n : (X → X)  • 𝟙 {ℓ₀}))
  ≡⟨ "Next symbol is not a ‘Π’, so Π→λ stops" ⟩'
   λ (X : Set) → λ (s : X) → Σ n : (X → X)  • 𝟙 {ℓ₀}
```

For pragmatism, we define a macro `_:waist_` such that ρ `:waist n` ≡ Π→λ (ρ `n`). Were we to attempt to prove such an equation in Agda, supposing, say, ρ : ℕ → `Set` and `n` : ℕ, by definition chasing (i.e., normalisation) the left side would immediatly reduce to ρ whereas the right side would reduce to ρ `n`; resulting in two distinct expressions. However, by inspecting the definitions, the only difference between the two is in the first line: Π→λ takes an instantiated context, whereas `_:waist_` takes a context and a ‘waist integer’ to instantiate the given context.

Waist

```
ρ :waist n = Π→λ (ρ n)
```

Source

```
{- ρ :waist n  ≡  Π→λ (ρ n)  -}
macro
  _:waist_ : (pkg : Term) (height : Term) (goal : Term) → TC Unit.⊤
  _:waist_ pkg n goal = normalise (pkg app n)
                        >>=term λ ρ → checkType goal (Π→λ-type ρ)
                        >>=term λ _ → unify goal (Π→λ-helper ρ)
```

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 ⟨ … ⟩, as defined below.

Syntactic Sugar for Context Values

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

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

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

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

Unbundling: Lifting Fields into Parameters

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

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

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

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

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

Examples of the need for such on-the-fly unbundling can be found in numerous places in the Haskell standard library. For instance, the standard libraries[26] 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:

[26] *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)

Monoids without commitment

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

With this context, `(Monoid ℓ₀ :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[27] useful for ad-hoc polymorphism.

7.4. Termtypes as Fixed-points

We have a practical monadic syntax for possibly parameterised record types that we would like to extend to termtypes. As discussed in the previous section, we could alter the bind operator to account for $\mathcal{W}$-types, but we shall present a different technique so as to avoid "making bind do too much". Algebraic data types are a means to declare concrete representations of the least fixed-point of a functor; see Swierstra[28] for more on this idea. In particular, the description language $\mathbb{D}$ for dynamical systems, below, declares concrete constructors for a fixpoint of a certain functor $\mathcal{D}$; i.e., $\mathbb{D} \cong$ `Fix` $\mathcal{D}$ where:

ADTs and Functors

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

𝒟 : Set → Set
𝒟 = λ (D : Set) → 𝟙 ⊎ D

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

The problem is whether we can derive $\mathcal{D}$ from `DynamicSystem`. Let us attempt a quick calcu-

[27] 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

[28] 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

lation sketching the necessary transformation steps (informally expressed via "⇝"):

From Contexts to Fixed-points: A Roadmap

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

Since we may view an algebraic data-type as a fixed-point of the functor obtained from the union of the sources of its constructors, it suffices to treat the fields of a record as constructors, then obtain their sources, then union them. That is, since algebraic-datatype constructors necessarily target the declared type, they are determined by their sources. For example, considered as a unary constructor `op : A → B` targets the termtype `B` and so its source is `A`. Hence, we can form the `termtype` of a context as the `Fix`-point of the sum —using Σ→⊎— of the `sources` of the context, as shown below. Where the operation Σ→⊎ rewrites dependent-sums into disjoint sums, which requires the second argument to lose its reference to the first argument which is accomplished by ↓↓; further details can be found in the appendices.

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

↓↓ τ = "reduce all de-bruijn indices within τ by 1"

Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (↓↓ Ba)

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

Before moving to an instructive **use** of this combinator, let us touch a bit on the details of its **formation**.

7.4.1. The `termtype` combinator

Using the guiding calculation above, we shall work up to the desired functor $\mathcal{D}$ by *implementing* each stage i of the calculation and showing the approximation $\mathtt{D}_i$ of the functor $\mathcal{D}$ at that stage.

1. Stage 1: Records The first step is already possible, using the existing `Context` setup.

Building up to the `termtype` combinator

```
D₁ = DynamicSystem 0

1-records : D₁ ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • 𝟙 {ℓ₀})
1-records = refl
```

2. Stage 2: Parameterised Records The second step is also already implemented, using the existing `_:waist_` mechanism.

Building up to the `termtype` combinator

```
D₂ = DynamicSystem :waist 1

2-funcs : D₂ ≡ (λ (X : Set) → Σ z : X • Σ s : (X → X) • 𝟙 {ℓ₀})
2-funcs = refl
```

3. Stage 3: Sources

As per the informal description of `sources` in the guiding calculation, we reinforce the idea with a number of desired test cases —as usual, formal machine checked test cases and Agda code can be found on the thesis repository. In particular, we make a **design decision** for the resulting `termtype` combinator: Types starting with implicit arguments are *invariants*, not *constructors* —and so are dropped from the resulting ADT by replacing them with the empty type '𝟘'.

Example uses of `sources`

τ	sources τ
`Src → Tgt`	`Src`
`Σ f : (Src → Tgt) • Bdy`	`Σ x : Src • Bdy`
`τ₁ → ⋯ → τₙ`	`τ₁ × ⋯ × τₙ₋₁ × 𝟙`
`Σ f : τ₁ → ⋯ → τₙ • Bdy`	`Σ x : (τ₁ × ⋯ × τₙ₋₁) • Bdy`
`∀ {x : ℕ} → x ≡ x`	`𝟘`
`(∀ {x y z : ℕ} → x ≡ y)`	`𝟘`
`𝟙`	`𝟘`

The third stage can now be formed.

Building up to the `termtype` combinator

```
D₃ = sources D₂

3-sources : D₃ ≡ λ (X : Set) → Σ z : 𝟙 • Σ s : X • 𝟘
3-sources = refl
```

With the following definitions.

sources

```
sourcesₜ (Π a : A • Ba) = A
sources (ℬ x : (Π a : A • Ba) • τ) = (ℬ x : A •
sources τ)
sources (ℬ x : A          • τ) = (ℬ x : 𝟙 • sources τ)
```

Where $\mathcal{B}$ is one of the binders λ or Σ.

Building up to the `termtype` combinator

```
-- The source of a type, not an arbitrary term.
-- E.g., sources (Σ x : τ • body) = Σ x : sourcesₜ τ • sources body
sourcesₜ : Term → Term

{- “Π {a : A} • Ba” ↦ 𝟘 -}
sourcesₜ (pi (arg (arg-info hidden _) A) _) = quoteTerm 𝟘

{-  “Π a : A • Π b : Ba • C a b” ↦ “Σ a : A • Σ b : B a • sourcesₜ (C a
↪ b)” -}
sourcesₜ (pi (arg a A) (abs ‘‘a’’ (pi (arg b Ba) (abs ‘‘b’’ Cab)))) =
  def (quote Σ) (vArg A
                :: vArg (lam visible (abs ‘‘a’’
                   (def (quote Σ)
                        (vArg Ba
                       :: vArg (lam visible (abs ‘‘b’’ (sourcesₜ Cab)))
                       :: []))))
                :: [])

{-  “Π a : A • Ba” ↦ “A” provided Ba does not begin with a Π  -}
sourcesₜ (pi (arg a A) (abs ‘‘a’’ Ba)) = A

{- All other non function types have an empty source; since X ≅ (𝟙 → X) -}
sourcesₜ _ = quoteTerm (𝟙 {ℓ₀})

{-# TERMINATING #-} -- Termination via structural smaller arguments is not
↪ clear due to the call to List.map
sources_term : Term → Term

sources_term (pi a b) = sourcesₜ (pi a b)
{- “Σ x : τ • Bx” ↦  “Σ x : sourcesₜ τ • sources Bx” -}
sources_term (def (quote Σ) (ℓ₁ :: ℓ₂ :: τ :: body))
    = def (quote Σ) (ℓ₁ :: ℓ₂ :: map-Arg sourcesₜ τ :: List.map (map-Arg
    ↪ sources_term) body)

{- This function introduces 𝟙s, so let's drop any old occurances a la 𝟘. -}
sources_term (def (quote 𝟙) _) = def (quote 𝟘) []

-- TODO: Maybe we do not need these cases.
sources_term (lam v (abs s x))     = lam v (abs s (sources_term x))
sources_term (var x args) = var x (List.map (map-Arg sources_term) args)
sources_term (con c args) = con c (List.map (map-Arg sources_term) args)
sources_term (def f args) = def f (List.map (map-Arg sources_term) args)
sources_term (pat-lam cs args) =  pat-lam cs (List.map (map-Arg sources_term)
↪ args)

-- sort, lit, meta, unknown
sources_term t = t

macro
  sources : Term → Term → TC Unit.⊤
  sources tm goal = normalise tm >>=_term λ tm' → unify (sources_term tm')
  ↪ goal
```

4. Stage 4: $\Sigma{\to}\uplus$ –Replacing Products with Sums

   As another tersely introduced utility, let us flesh-out $\Sigma{\to}\uplus$ by means of a few desired unit tests —notice that the final example concerns a parameterised dynamical system. As mentioned in the guiding calculation, we will replace unit types by empty types —i.e., "empty Σ-products by empty ⊎-sums".

τ	$\Sigma{\to}\uplus\ \tau$
`Π S : Set • (S → S)`	`Π S : Set • (S → S)`
`Π S : Set • Σ n : S • S`	`Π S : Set • S ⊎ S)`
`Π S : Set • Σ n : (S → S) • S`	`Π S : Set • (S → S) ⊎ S)`
`λ S : Set • Σ s : S • Σ n : (S → S) • 𝟙`	`λ S : Set • S ⊎ (S → S) ⊎ 𝟘`

   Decreasing de Brujin Indices: Any given quantification `(Σ x : τ • fx)` may have its body `fx` refer to the free variable `x`. If we decrement all de Bruijn indices `fx` contains, then there would be no reference to `x`. (In the repository code, ↓↓ appears as `var-dec`.)

⇊

Building up to the `termtype` combinator

```
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 (pi X (abs b Bx)) = 1 + lengthₜ Bx
{-# CATCHALL #-}
-- sort, lit, meta, unknown
lengthₜ t = 0
-- The Length of a Term:1 ends here

-- [[The Length of a Term][The Length of a Term:2]]
_ : lengthₜ (quoteTerm (Σ x : ℕ • x ≡ x)) ≡ 10
_ = refl

--
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 𝟘) []
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) (pi (arg a A) (abs b Ba)) = pi (arg a (var-dec₀ n A)) (abs
↪  b (var-dec₀ n Ba))
-- 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

var-dec : Term → Term
var-dec t = var-dec₀ (lengthₜ t) t
```

Notice that we made the decision that `x`, in the body of `(Σ x • x)`, will reduce to 𝟘, the empty type. Indeed, in such a situation the only Debrujin index cannot be reduced further; e.g., `⇊(quoteTerm x) ≡ quoteTerm ⊥`.

Σ→⊎

var-dec τ = "reduce all de-bruijn indices within τ by 1"
Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (var-dec Ba)
Σ→⊎ ($\mathcal{B}$ a : A • Ba) = ($\mathcal{B}$ a : A • Σ→⊎ Ba) for other binders $\mathcal{B}$, such as Π or λ.

Building up to the `termtype` combinator

```
{-# TERMINATING #-}
Σ→⊎₀ : Term → Term

{- "Σ a : A • Ba" ↦ "A ⊎ B" where 'B' is 'Ba' with no reference to 'a'
↪  -}
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.
-- See the unit tests above ;-)
-- For some reason, the inclusion of this caluse obscures structural
↪  termination.
Σ→⊎₀ (def (quote 𝟙) _) = def (quote 𝟘) []

 -- Walk under λ's and Π's.
Σ→⊎₀ (lam v (abs s x)) = lam v (abs s (Σ→⊎₀ x))
Σ→⊎₀ (pi A (abs a Ba)) = pi A (abs a (Σ→⊎₀ Ba))
Σ→⊎₀ t = t

macro
  Σ→⊎ : Term → Term → TC Unit.⊤
  Σ→⊎ tm goal = normalise tm >>=term λ tm' → unify (Σ→⊎₀ tm') goal
```

We can now form the fourth stage approximation of the functor $\mathcal{D}$; in-fact we will use this form as *the definition* of the desired functor $\mathcal{D}$ —since the sum with 𝟘 *essentially* contributes nothing.

Building up to the `termtype` combinator

```
D₄ = Σ→⊎ D₃

4-unions : D₄ ≡ λ X → 𝟙 ⊎ X ⊎ 𝟘
4-unions = refl
```

5. Stage 5: Fixpoint Since we want to define algebraic data-types as fixed-points, we are led inexorably to using a recursive type that fails to be positive.

Building up to the `termtype` combinator

```
{-# NO_POSITIVITY_CHECK #-}
data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
  μ : F (Fix F) → Fix F
```

Building up to the `termtype` combinator

```
𝔻 = Fix D₄
```

We summarise the stages together into one macro:

Termtype

```
termtype : UnaryFunctor → Type
termtype τ = Fix (Σ→⊎ (sources τ))
```

Building up to the `termtype` combinator

```
macro
  termtype : Term → Term → TC Unit.⊤
  termtype tm goal =
                normalise tm
          >>=term λ tm' → unify goal (def (quote Fix) ((vArg (Σ→⊎₀
          ↪  (sourcesterm tm'))) :: []))
```

Then, we may instead declare:

Building up to the `termtype` combinator

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

7.4.2. Instructive Example: $\mathbb{D} \cong \mathbb{N}$

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

Declaring a Derived Termtype

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

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

With these `pattern` declarations, we can actually use the more meaningful names `startD` and `nextD` when pattern matching, instead of the seemingly daunting 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)
```

Seemingly Trivial Remappings

```
module free-dynamical-system where

    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

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

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

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

```
Inj n x = μ (inj₂ ⁿ (inj₁ x))
```

Seemingly Trivial Remappings

```
-- i-th injection:  (inj₂ ∘ ⋯ ∘ inj₂) ∘ inj₁
Inj₀ : ℕ → Term → Term
Inj₀ zero c    = con (quote inj₁) (arg (arg-info visible relevant) c ∷ [])
Inj₀ (suc n) c = con (quote inj₂) (vArg (Inj₀ n c) ∷ [])

macro
  Inj : ℕ → Term → Term → TC Unit.⊤
  Inj n t goal = unify goal ((con (quote μ) []) app (Inj₀ n t))
```

we may define `startD = Inj 0 tt` and `nextD e = Inj 1 e`—that is, constructors of termtypes are particular injections into the possible summands that the termtype consists of.

7.5. Free Datatypes from Theories

Astonishingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if a parameterised context `𝒞 : Set → Context ℓ₀` is given, then `ℂ = λ X → termtype (𝒞 X :waist 1)` can be used to form 'free, lawless, 𝒞-instances'. For instance, earlier we witnessed that the termtype of dynamical systems is essentially the natural numbers.

Data structures as free theories

Theory	Termtype
Dynamical Systems	ℕ
Pointed Structures	Maybe
Monoids	Binary Trees

The final entry in the above table is a well known correspondence that we can now not only formally express, but also prove to be true. As we did with dynamical systems, we begin with forming 𝕄 the termtype of monoids, then using `pattern` clauses to provide compact names, and explicitly form the algebraic `data` type of trees.

Trees from Monoids

```
𝕄 : Set
𝕄 = termtype (Monoid ℓ₀ :waist 1)

that-is : 𝕄 ≡ Fix (λ X → X × X × 𝟙 -- _⊕_, branch
                    ⊎ 𝟙          -- Id, nil leaf
                    ⊎ 𝟘          -- invariant leftId
                    ⊎ 𝟘          -- invariant rightId
                    ⊎ 𝟘          -- invariant assoc
                    ⊎ 𝟘)         --  the "End {ℓ}"
that-is = refl

-- 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

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

to∘from : ∀ t → to (from t) ≡ t
to∘from empty        = refl
to∘from (branch l r) = cong₂ branch (to∘from l) (to∘from 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.

7.6. Language Agnostic Construction

In contrast to the generic approach to semantics for contexts of section 5.4, here we generalise the previous setup to an arbitrary Generalised Type Theory —as defined in Chapter 2, and used to place the prototype on solid foundations. We present a quick sketch —and so *omit* the full typing rules of the claimed operators, leaving that as an exercise for the interested reader (some of which are already present in Chapter 2).

Suppose we have a language consisting of 'terms' and a typing relation '⊢'. To implement the `Context` library for such a language, we need to have access to 3 classes of constructions:

1. Dependent function types `Π a : A • Ba` with values `λ a : A • ba` and the usual function application eliminator —where `A : Type` and `a : A ⊢ Ba : Type` and `a : A ⊢ ba :`

Ba— and there is a unit and type ⊢ 𝟙 : Type and the natural numbers ⊢ ℕ : Type.

2. Dependent record types Σ a : A • Ba —where A : Type and a : A ⊢ Ba : Type— and there is an empty type and ⊢ 𝟘 : Type.

3. An operator Fix that maps *polynomial functors* to their initial algebras —notice that it does not need to be a generic fixpoint operator.

We have 3 classes corresponding to the 3 primitive ways to view a context —Π, Σ, and $\mathcal{W}$ as discussed in Chapter 2. The more of these features that a language has, the more of the Context system it can implement.

Π; λ; ℕ; 𝟙 ⇒ Context

```
-- ⊢ Context : Type
Context  =  Π _ : ℕ • Type

-- ⊢ End : Context
End      =  λ _ : ℕ • 𝟙
```

Σ ⇒ ⟫=

```
-- ⊢ _⟫=_ : Π Γ : Type • Π _ : (Π _ : Γ • Context) • Context
(Γ ⟫= f) 0        =  Σ γ : Γ • f γ 0
(Γ ⟫= f) (n + 1)  =  Π γ : Γ • f γ n
```

𝟘 ⇒ Typeclasses with Π→λ

```
-- ⊢ Π→λ : Π A : Type • Π _ : Type • Π _ : A • Type
Π→λ A (Π a : A • τ)  =  λ a : A • Π→λ τ
Π→λ A _              =  λ a : A • 𝟘

-- ⊢ :waist : Π A : Type • Π _ : Context • Π _ : ℕ • Π _ : A • Type
:waist A ρ n  =  Π→λ (ρ n)
```

The final piece, regarding termtypes, requires a mechanism provided for forming guarded definitions —in Agda this is accomplished with the with keyword.

Fixpoints ⇒ $\mathcal{W}$-types

```
-- ⊢ sources : Π _ : (Π _ : Type • Type) • Π _ : Type • Type
sources (λ x : (Π a : A • Ba) • τ) = λ x : A • sources τ
sources (λ x : A               • τ) = λ x : 𝟙 • sources τ
sources _                            = λ x : 𝟘 • 𝟘

-- ⊢ Σ→⊎ : Π _ : Type • Type
Σ→⊎ (Σ a : A • B)  =  A ⊎ Σ→⊎ B   provided  ⊢ B : Type
Σ→⊎ _              =  𝟘

-- ⊢ termtype : Π τ : (Π _ : Type • Type) • Type
termtype τ = Fix (Σ→⊎ (sources τ))
```

Since ⟫= ensures that `Context` values are always formed from sums Σ and products Π, we have polynomial constructions and so it suffices to find the initial algebra of such operators —which always exist; see section 5.3 on $\mathcal{W}$-types. We assumed `Fix` yields such algebras.

7.7. Conclusion

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

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

Contexts embody all kinds of grouping mechanisms

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

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

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

8. Conclusion

The initial goal of this work was to explore how investigations into packaging-up-data —and language extension in general— could benefit from mechanising tedious patterns, thereby reinvigorating the position of universal algebra within computing. Towards that goal, we have decided to create an editor extension that can be used, for instance, to quickly introduce universal algebra constructions for the purposes of "getting things done" in a way that does not force users of an interface to depend on features they do not care about —the so-called Interface Segregation Principle. Moreover, we have repositioned the prototype from being an auxiliary editor extension to instead being an in-language library and have presented its key insights so that can be developed in other dependently-typed settings besides Agda.

Based on the results —such as the 750% line savings in the MathScheme library— we are convinced that the (one-line) specification of common theories (data-structures) can indeed be used to reinvigorate the position of universal algebra in computing, as far as DTLs are concerned. The focus on the modular nature of algebraic structures, for example, allows for the *mechanical* construction of novel and unexpected structures in a practical and elegant way —for instance, using the `keeping` combinator to extract the *minimal* interface for an operation, or proof, to be valid. Also, we believe that the correspondence between abstract mathematical theories and data structures in computing only strengthens the need for a mechanised approach for the under-utilised constructions available on the the mathematical side of the correspondence.

Some preliminary experiences show that the approach used in this thesis can be used with immediate success. For example, the editor extension allows a host of renamings to be done, along with the relevant relationship mappings, and so allow proofs to be written in a more readable fashion. As another example, the in-language library allows one to show that the free algebra associated with a theory is a particular useful and practical data-structure —such as `ℕ`, `Maybe`, and `List`. These two examples are more than encouraging, for the continual of this effort. Also, the success claimed by related work like Arend[30,31] makes us believe that we can have a positive impact.

This thesis has focused on various aspects of furnishing packages with a status resemebling that of a first-class citizen in a dependently-typed language. Where possible, we will give an indication of future work which has still to be done to get more insight in this direction.

[30] JetBrains Research. *Arend Theorem Prover*. 2020. URL: https://arend-lang.github.io/

[31] Valery Isaev. "Models of Homotopy Type Theory with an Interval Type". In: *CoRR* abs/2004.14195 (2020). arXiv: 2004.14195. URL: https://arxiv.org/abs/2004.14195

Chapter Contents

8.1. Questions, Old and New

Herein we revisit the research questions posed in the introductory chapter, summarise our solutions to each, and discuss future work.

Practical Concern ♯1: Renaming & Remembering Relationships. A given structure may naturally give rise to various 'children structures', such as by adding-new/dropping-old/renaming componenets, and it is useful to have a (possibly non-symmetric) coercision between the child and the original parent.

We have succeeded to demonstrate that ubquitious constructions can be mechanised and the coercisions can also be requested by a simple keyword in the specification of the child structure. As far as this particular problem is concerned, we see no missing feature and are content with the success that the PackageFormer prototype has achieved. However, the in-language Context library does leave room for improvement, but this is a limitation of the current Agda reflection mechanism rather than of the approach outlined by PackageFormer.

Practical Concern ♯2: Unbundling. A given structure may need to have some of its components 'fixed ahead of time'. For instance, if we have a type `Graph` of graphs but we happen to be discussing only graphs with natural numbers as nodes, then we need to work with `Σ G : Graph • G.Node ≡ ℕ` and so work with pairs `(G, refl)` whose second component is a necessarily technical burden, but is otherwise insightful.

Our framework(s) achieve this goal, joyously so. An improvement would be not to blindly lift the first n-many componenets to the type level but instead to expose the induced depenencdy subgraph of a given set of componenets. The PackageFormer already does this for the `keeping` combinator and the same code could be altered for the `waist` combinator. At first, it would seem that a similar idea would work for the in-language library, however this is not the case. The Context library, unlike PackageFormer, does not work with flat strings but instead transforms the inner nodes of abstract syntax trees —such as replacing Πs by λs or Σs— and so the need to lift a subgraph of a structure's signature no longer becomes a linear operation that alters inner nodes.

Perhaps an example would illuminate the problem. Consider the following signature.

PSGwId^2 —'P'ointed 'S'emi'g'roup 'w'ith 'Id^2' ≈ Id

```
record PSGwId² : Set₁ where
  field
    -- We have a semigroup
    C     : Set
    _⊕_   : C → C → C
    assoc : ∀ x y z  →  (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
    -- with a selected point
    id    : C

  twice : C → C
  twice = λ x → x ⊕ x

  -- Such that the point is idempotent
  field
    id² : twice id ≡ id
```

Suppose we want to have the field `id²` at the type level, then we must also expose the parts of the signature that make it well-defined; namely, `C, _⊕_, id, twice`. At a first pass, `id²` only needs `id` and the operation `twice`; however, if we look at each of these in-turn we see that we also need `C` and `_⊕_`. As such, in the worst case, this operation is quadratic. Moving on, as the signature is traversed, we can mark fields to be lifted but we need a combinator to "shift leftward (upward)" the names that are to be at the type level —in this case, we need to move `id²` and `id` to come before `assoc`. This is essentially the algorithm implemented in PackageFormer's `keeping` combinator. However, for Context's `do`-notation, this may not be possible since inner-nodes are no longer replaced, linearly, according to a single toggle. Furture work would be to investigate whether it would be possible and, if so, how to do so in a *pragmattic and usable* fashion.

Theoretical Concern ♯1: Exceptionality. If an integer m divides an integer n, then division nm yields an integer witnessing n as a multiple of m; likewise, if a package p is structurally (nominally) contained in a package q, then we can form a pacakge, say, $q \mathbin{\dot{-}} p$ that contains the extra matter and it is parameterised by an instance of p —e.g., `Monoid` is contained in `Group` and so `Group ∸ Monoid = λ (M : Monoid) → (_⁻¹ : ···, left-inverse : ···, right-inverse: ···)` is the paramterised pacakage that can adjoin inverses to monoids. As such, packages are like numbers —compare with the idea that a list is like a number, the latter being a list of unit (trivial) information.

Our goal was to determined the *feasiblility* of this idea *within* dependently-typed settings. The implementation of the the Context in-language library yields a resounding positive. As mentioned already, limitations of the host DTL's reflection mechanism are inherited by our approach.

Furture work would focus on the precise relationship between features of the host language and a library treating packages as first-class. Moreover, it would be useful to investigate how packages can be promoted to first-class *after* the construction of a language. Such an investigation would bring to light the interplay of how packages actually influence other parts of a language —which is sorely lacking from our work.

Perhaps the most pressing concern would be how the promotion of packages would influence typechecking. At first, for instance, the package `PSGwId`^{2} from above could be typed as $\texttt{Set}_1$ but that would be wildely inappropriate since we cannot apply arbitrary package combinators, such as `_÷_`, to arbitrary types —just as we cannot apply `_÷_` to arbitrary types. Instead, we would need a dedicated type, say, `Package`. Things now become exceedingly hairy. Do we need a hierarchy or avoid paradoxes, as is the case with $\texttt{Set}_n$? A parameterised type is a Π-type, but a paramterised package *is a* package —so do Π-types get 'absorped' into `Package`? What are the types of the package combinators introduced in this thesis, such as unbundliungs $\Pi{\rightarrow}\lambda$?

These questions are not only interesting by themselves but they would also be a stepping stone in having full-fledged first-class pacakages in dependently-typed languages.

Theoretical Concern ♯2: Syntax. The theories-as-data-structures lens presented in this work showcases how a theory (a record type, signature, admitting instances) can have useful data-structures (algebraic data types) associated with it. For instance, monoids give rise to binary trees whose leaf values are drawn from a given carrier (variable) set. One can then encode a sentence of a model structure using the syntax, perform a syntactic optimisation, then interpret the sentence using the given instance.

We are delighted with the rather unexpected success of this aspect of our work. The formal methods community is well-aware that monoids are related to binary trees and that pointed sets are related to maybe (nullable) types, yet we have had the honour of being the first to actually derive the latter from the former *mechanically*.

Future work would focus on the treatment non-function-symbols. For instance, instead of discarding properties from a theory, one could keep them thereby obtaining 'higher-order datatypes'[32] or could have them lifted as parameters in a (mechanically generated) subsequent module. Moreover, the current implementation of Context has a basic predicate determining what consitutes a function-symbol, it would be interesting to make that a parameter of the theories-as-data-structures `termtype` construction.

Proof. Finally, there are essentially no formal theorems proven in this work. The constructions presented rely on *typechecking*: One can phrase a desired construction and typechecking determines whether it is meaningful or not. It would be useful to determine the necessary conditions that guarantee the well-definedness of the constructions —so that we may then "go up another level" and produce meta-constructions that invoke our current constructions mechanically and "wholesale".

⇒ Actually, proof-checking is a part of type-checking since all proofs are terms.

[32] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical agda: a dependently typed programming language with univalence and higher inductive types". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 87:1–87:29. DOI: 10.1145/3341691. URL: https://doi.org/10.1145/3341691

⇒ Reformulate this paragraph to make it clear what is proven and how via typechecking. Eg typechecking are examples and the more general meahcnisms and this difference matters.

8.2. Concluding Remarks

In dependently-typed settings (DTS), it is common practice to operate on packages —by renaming them, hiding parts, adding new parts, etc.— and the frameworks presented in this thesis show that it is indeed possible to treat packages nearly as first-class citizens "after the fact" even when a language does not assign them such a status. The techniques presented show that this approach is feasible as an in-language library for DTS as well as for the any highly customisable and extensible text editor.

The combinators presented in this thesis were guided not by theoretical concerns on the algebraic nature of containers but rather on the practical needs of actual users working in DTS. We legitimately believe that that our stance on packages as first-class citizens should —and hopefully one day would— be an integral part of any DTS. The Context library is a promising approach to promoting the status of packages, to reducing the gap between different "sub-languages" in a language, and allowing users to benefit from a streamlined and familiar approach to packages —as if they were the 'fancy numbers' abstracted by rings, fields, and vector spaces.

Finally, even though we personally believe in the import of packages, we do not expect the same belief to trickle-down to mainstream languages immediately since they usually do not have sufficiently sophisticated[33] type systems to permit the treatment of packages as first-class citizens, on the same footing as numbers. Nonetheless, we believe that the work in this thesis is yet another stepping-stone on the road of *DRY*[34] endeavours.

[33] The static typing of some languages, such as C, is so pitiful that is makes type systems seem more like a burden then anything useful —in C, one often uses void pointers to side-step the type system's limitations, thereby essentially going untyped. The dynamically typed languages, however, could be an immediate test-bed for package combinators —indeed, Lisp, Python, and JavaScript use 'splicing' operators to wholesale include structures in other structures, within the core language.

[34] *Don't Repeat Yourself!*

Bibliography

[1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. “Representing Nested Inductive Types Using W-Types”. In: *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings.* Ed. by Josep Díaz et al. Vol. 3142. Lecture Notes in Computer Science. Springer, 2004, pp. 59–71. ISBN: 3-540-22849-7. DOI: `10.1007/978-3-540-27836-8\_8`. URL: `https://doi.org/10.1007/978-3-540-27836-8%5C_8`.

[2] Andreas Abel and Gabriel Scherer. “On Irrelevance and Algorithmic Equality in Predicative Type Theory”. In: *Log. Methods Comput. Sci.* 8.1 (2012). DOI: `10.2168/LMCS-8(1:29)2012`.

[3] S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, eds. *Handbook of Logic in Computer Science: Volume 5. Algebraic and Logical Structures*. Oxford University Press, Jan. 2001. DOI: `10.1093/oso/9780198537816.001.0001`. URL: `https://doi.org/10.1093%2Foso%2F9780198537816.001.0001`.

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

[5] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. “Observational equality, now!” In: *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*. 2007, pp. 57–68. DOI: `10.1145/1292597.1292608`.

[6] Thorsten Altenkirch et al. “Indexed containers”. In: *J. Funct. Program.* 25 (2015). DOI: `10.1017/S095679681500009X`. URL: `https://doi.org/10.1017/S095679681500009X`.

[7] 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`.

[8] Roland Carl Backhouse and Paul Chisholm. “Do-It-Yourself Type Theory”. In: *Formal Aspects Comput.* 1.1 (1989), pp. 19–84. DOI: `10.1007/BF01887198`. URL: `https://doi.org/10.1007/BF01887198`.

[9] John C. Baez and Michael Shulman. *Lectures on n-Categories and Cohomology*. 2006. arXiv: `math/0608420v2 [math.CT]`.

[10] Clemens Ballarin. “Locales and Locale Expressions in Isabelle/Isar”. In: *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*. 2003, pp. 34–50. DOI: `10.1007/978-3-540-24849-1\_3`. URL: `https://doi.org/10.1007/978-3-540-24849-1%5C_3`.

[11] Andrej Bauer. “Five stages of accepting constructive mathematics”. In: *Bulletin of the American Mathematical Society* (2016). DOI: https://doi.org/10.1090/bull/1556.

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

[13] 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.

[14] Ana Bove, Peter Dybjer, and Ulf Norell. “A Brief Overview of Agda — A Functional Language with Dependent Types”. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17–20, 2009. Proceedings*. 2009, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6.

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

[16] Stanley Burris and Hanamantagouda P. Sankappanavar. *A course in universal algebra*. Vol. 78. Graduate texts in mathematics. Springer, 1981. ISBN: 978-0-387-90578-5.

[17] Jacques Carette, William M. Farmer, and Michael Kohlhase. *Realms: A Structure for Consolidating Knowledge about Mathematical Theories*. 2014. arXiv: 1405.5956v1 [cs.MS].

[18] 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.

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

[20] John Cartmell. “Generalised algebraic theories and contextual categories”. In: *Ann. Pure Appl. Log.* 32 (1986), pp. 209–243. DOI: 10.1016/0168-0072(86)90053-9. URL: https://doi.org/10.1016/0168-0072(86)90053-9.

[21] R. I. Chaplin, R. E. Crosbie, and J. L. Hay. “A Graphical Representation of the Backus-Naur Form”. In: *Comput. J.* 16.1 (1973), pp. 28–29. DOI: 10.1093/comjnl/16.1.28. URL: https://doi.org/10.1093/comjnl/16.1.28.

[22] Noam Chomsky. “A Note on Phrase Structure Grammars”. In: *Inf. Control.* 2.4 (1959), pp. 393–395. DOI: 10.1016/S0019-9958(59)80017-6. URL: https://doi.org/10.1016/S0019-9958(59)80017-6.

[23] Noam Chomsky. “On Certain Formal Properties of Grammars”. In: *Inf. Control.* 2.2 (1959), pp. 137–167. DOI: 10.1016/S0019-9958(59)90362-6. URL: https://doi.org/10.1016/S0019-9958(59)90362-6.

[24] Emily Cibelli et al. “The Sapir-Whorf Hypothesis and Probabilistic Inference: Evidence from the Domain of Color”. In: *Proceedings of the 38th Annual Meeting of the Cognitive Science Society, Recogbizing and Representing Events, CogSci 2016, Philadelphia, PA, USA, August 10-13, 2016*. Ed. by Anna Papafragou et al. cognitivesciencesociety.org, 2016. ISBN: 978-0-9911967-3-9. URL: https://mindmodeling.org/cogsci2016/papers/0493/index.html.

[25] Manuel Clavel et al., eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-71940-3. DOI: 10.1007/978-3-540-71999-1. URL: https://doi.org/10.1007/978-3-540-71999-1.

[26] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*. Apr. 2018. DOI: 10.5281/zenodo.1219885. URL: https://hal.inria.fr/hal-01954564.

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

[28] Michael Dante DiMartino and Bryan Konietzko. *Avatar, the last airbender*. Premiered on Nickelodeon. 2005.

[29] Francisco Durán and José Meseguer. "Maude's module algebra". In: *Sci. Comput. Program.* 66.2 (2007), pp. 125–153. DOI: 10.1016/j.scico.2006.07.002. URL: https://doi.org/10.1016/j.scico.2006.07.002.

[30] Peter Dybjer. "Representing inductively defined sets by wellorderings in Martin-Löf's type theory". In: *Theoretical Computer Science* 176.1-2 (Apr. 1997), pp. 329–335. ISSN: 0304-3975. DOI: 10.1016/s0304-3975(96)00145-4. URL: http://dx.doi.org/10.1016/s0304-3975(96)00145-4.

[31] Elliott. "Denotational design with type class morphisms". In: 2016. URL: http://conal.net/papers/type-class-morphisms/type-class-morphisms-long.pdf.

[32] Jacopo Emmenegger. *W-types in setoids*. 2018. arXiv: 1809.02375v2 [math.LO].

[33] William M. Farmer. "A New Style of Mathematical Proof". In: *Mathematical Software - ICMS 2018 - 6th International Conference, South Bend, IN, USA, July 24-27, 2018, Proceedings*. Ed. by James H. Davenport et al. Vol. 10931. Lecture Notes in Computer Science. Springer, 2018, pp. 175–181. ISBN: 978-3-319-96417-1. DOI: 10.1007/978-3-319-96418-8_21. URL: https://doi.org/10.1007/978-3-319-96418-8%5C_21.

[34] William M. Farmer. *A New Style of Proof for Mathematics Organized as a Network of Axiomatic Theories*. 2018. arXiv: 1806.00810v2 [cs.LO].

[35] 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.

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

[37] Nicola Gambino and Martin Hyland. "Wellfounded Trees and Dependent Polynomial Functors". In: *Types for Proofs and Programs* (2004), pp. 210–225. ISSN: 1611-3349. DOI: 10.1007/978-3-540-24849-1_14. URL: http://dx.doi.org/10.1007/978-3-540-24849-1_14.

[38] 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.

[39] 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.

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

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

[42] Jason Gross, Adam Chlipala, and David I. Spivak. *Experience Implementing a Performant Category-Theory Library in Coq*. 2014. arXiv: 1401.7694v2 [math.CT].

[43] Guoyong, Peimin Deng, and Jiali Feng. "Specification based on Backus-Naur Formalism and Programming Language". In: *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 - December 1, 2002, Proceedings*. 2002, pp. 95–101.

[44] *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).

[45] 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.

[46] Jaakko Hintikka and Merrill B. Hintikka. "On Denoting what?" In: *The Logic of Epistemology and the Epistemology of Logic: Selected Essays*. Dordrecht: Springer Netherlands, 1989, pp. 165–181. ISBN: 978-94-009-2647-9. DOI: 10.1007/978-94-009-2647-9_11.

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

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

[49] 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.

[50] Valery Isaev. "Models of Homotopy Type Theory with an Interval Type". In: *CoRR* abs/2004.14195 (2020). arXiv: 2004.14195. URL: https://arxiv.org/abs/2004.14195.

[51] B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland, 1999.

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

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

[54] Wolfram Kahl and Jan Scheffczyk. "Named Instances for Haskell Type Classes". In: 2001.

[55] Anne Kaldewaij. *Programming - the derivation of algorithms*. Prentice Hall international series in computer science. Prentice Hall, 1990. ISBN: 978-0-13-204108-9.

[56] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. "Locales - A Sectioning Concept for Isabelle". In: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*. 1999, pp. 149–166. DOI: 10.1007/3-540-48256-3_11. URL: https://doi.org/10.1007/3-540-48256-3%5C_11.

[57] Donald E. Knuth. "backus normal form vs. Backus Naur form". In: *Commun. ACM* 7.12 (1964), pp. 735–736. DOI: 10.1145/355588.365140. URL: https://doi.org/10.1145/355588.365140.

[58] Jeroen F. J. Laros et al. "A formalized description of the standard human variant nomenclature in Extended Backus-Naur Form". In: *BMC Bioinform.* 12.S-4 (2011), S5. DOI: 10.1186/1471-2105-12-S4-S5. URL: https://doi.org/10.1186/1471-2105-12-S4-S5.

[59] Jan van Leeuwen, ed. *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0444880747.

[60] 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.

[61] 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.

[62] Gideon Makin. "Making sense of 'on denoting'". In: *Synth.* 102.3 (1995), pp. 383–412. DOI: 10.1007/BF01064122.

[63] 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.

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

[65] Conor McBride. “Dependently typed functional programs and their proofs”. PhD thesis. University of Edinburgh, UK, 2000. URL: http://hdl.handle.net/1842/374.

[66] 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.

[67] Alexander Mehler, Olga Pustylnikov, and Nils Diewald. “Geography of social ontologies: Testing a variant of the Sapir-Whorf Hypothesis in the context of Wikipedia”. In: *Comput. Speech Lang.* 25.3 (2011), pp. 716–740. DOI: 10.1016/j.csl.2010.05.006. URL: https://doi.org/10.1016/j.csl.2010.05.006.

[68] 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.

[69] Ulf Norell. “Towards a Practical Programming Language Based on Dependent Type Theory”. See also http://wiki.portal.chalmers.se/agda/pmwiki.php. PhD thesis. Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology, Sept. 2007.

[70] Anna Papafragou et al., eds. *Proceedings of the 38th Annual Meeting of the Cognitive Science Society, Recogbizing and Representing Events, CogSci 2016, Philadelphia, PA, USA, August 10-13, 2016*. cognitivesciencesociety.org, 2016. ISBN: 978-0-9911967-3-9. URL: https://mindmodeling.org/cogsci2016/.

[71] Christine Paulin-Mohring. “The Calculus of Inductive Definitions and its Implementation: the Coq Proof Assistant”. In: invited tutorial.

[72] Leonid I. Perlovsky. “Emotions, language, and Sapir-Whorf hypothesis”. In: *International Joint Conference on Neural Networks, IJCNN 2009, Atlanta, Georgia, USA, 14-19 June 2009*. IEEE Computer Society, 2009, pp. 2501–2508. ISBN: 978-1-4244-3548-7. DOI: 10.1109/IJCNN.2009.5178891. URL: https://doi.org/10.1109/IJCNN.2009.5178891.

[73] Frank Pfenning and The Twelf Team. *The Twelf Project*. 2015. URL: http://twelf.org/wiki/Main_Page (visited on 10/19/2018).

[74] Brigitte Pientka. “Beluga: Programming with Dependent Types, Contextual Data, and Contexts”. In: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*. 2010, pp. 1–12. DOI: 10.1007/978-3-642-12251-4_1. URL: https://doi.org/10.1007/978-3-642-12251-4%5C_1.

[75] Florian Rabe. “Representing Isabelle in LF”. In: *Electronic Proceedings in Theoretical Computer Science* 34 (Sept. 2010), pp. 85–99. ISSN: 2075-2180. DOI: 10.4204/eptcs.34.8. URL: http://dx.doi.org/10.4204/EPTCS.34.8.

[76] 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: https://doi.org/10.1145/1577824.1577831.

[77] JetBrains Research. *Arend Theorem Prover*. 2020. URL: https://arend-lang.github.io/.

[78] Bertrand Russell. “On Denoting”. In: *Mind* XIV.4 (Jan. 1905), pp. 479–493. ISSN: 0026-4423. DOI: 10.1093/mind/XIV.4.479. eprint: https://academic.oup.com/mind/article-pdf/XIV/4/479/9872659/479.pdf.

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

[80] Aaron Stump and David L. Dill. “Faster Proof Checking in the Edinburgh Logical Framework”. In: *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*. 2002, pp. 392–407. DOI: 10.1007/3-540-45620-1_32. URL: https://doi.org/10.1007/3-540-45620-1%5C_32.

[81] 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.

[82] Christine Tseng et al. “A computational investigation of the Sapir-Whorf hypothesis: The case of spatial relations”. In: *Proceedings of the 38th Annual Meeting of the Cognitive Science Society, Recogbizing and Representing Events, CogSci 2016, Philadelphia, PA, USA, August 10-13, 2016*. Ed. by Anna Papafragou et al. cognitivesciencesociety.org, 2016. ISBN: 978-0-9911967-3-9. URL: https://mindmodeling.org/cogsci2016/papers/0387/index.html.

[83] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[84] Christian Urban, James Cheney, and Stefan Berghofer. *Mechanizing the Metatheory of LF*. 2008. arXiv: 0804.1667v3 [cs.LO].

[85] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. “Cubical agda: a dependently typed programming language with univalence and higher inductive types”. In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 87:1–87:29. DOI: 10.1145/3341691. URL: https://doi.org/10.1145/3341691.

[86] Ed Voermans. “Pers as Types, Inductive Types and Types with Laws”. In: *Declarative Programming, Sasbachwalden 1991, PHOENIX Seminar and Workshop on Declarative Programming, Sasbachwalden, Black Forest, Germany, 18-22 November 1991*. 1991, pp. 274–291. DOI: 10.1007/978-1-4471-3794-8_18.

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

A. Code

Chapter Contents

A.1. 265 Line `Context` Implementation

```
--
-- The Next 700 Module Systems (•̀‿•́)و Musa Al-hassy ⟨2021-01-22
↪  Friday 16:25:19⟩
--
-- This file was mechanically generated from a literate program.
-- Namely, my PhD thesis on ‘do-it-yourself module systems for Agda’.
--
-- https://alhassy.github.io/next-700-module-systems/thesis.pdf
--
-- There are “[[backward][references]]” to the corresponding
↪  expository text.
--
-- Agda version 2.6.1.2; Standard library version 1.2

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)

import Data.Unit as Unit

-- The map-Args of Reflection is deprecated, and it is advised to use
↪  the map-Args
-- within Reflection.Argument.
open import Reflection hiding (name; Type; map-Arg;  map-Args)
↪  renaming (_>>=_ to _>>=ₜₑᵣₘ_)
open import Reflection.Argument using (map-Args) renaming (map to
↪  map-Arg)

ℓ₁   = Level.suc ℓ₀

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 𝟙 {ℓ} : Set ℓ where
  constructor tt

𝟘 = ⊥

-- [[Single argument application][Single argument application:1]]
_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
-- Single argument application:1 ends here

-- [[Reify ℕ term encodings as ℕ values][Reify ℕ term encodings as ℕ values:1]]
toℕ : Term → ℕ
toℕ (lit (nat n)) = n
{-# CATCHALL #-}
toℕ _ = 0
-- Reify ℕ term encodings as ℕ values:1 ends here

{- Type annotation -}
syntax has A a = a : A

has : ∀ {ℓ} (A : Set ℓ) (a : A) → A
has A a = a

-- From: https://alhassy.github.io/PathCat.html  § Imports
open import Relation.Binary.PropositionalEquality as ≡ using (_EQUAL_ ; _≡_)
module _ {i} {S : Set i} where
    open import Relation.Binary.Reasoning.Setoid (≡.setoid S) public

open import Agda.Builtin.String

defn-chasing : ∀ {i} {A : Set i} (x : A) → String → A → A
defn-chasing x reason supposedly-x-again = supposedly-x-again

syntax defn-chasing x reason xish = x ≡⟨ reason ⟩' xish

infixl 3 defn-chasing

{- “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 {ℓ} = ‘ 𝟙 {ℓ}

_>>=_ : ∀ {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

Π→λ-type : Term → Term
Π→λ-type (pi a (abs x b)) = pi a  (abs x (Π→λ-type b))
Π→λ-type x = unknown

Π→λ-helper : Term → Term
Π→λ-helper (pi a (abs x b)) = lam visible (abs x (Π→λ-helper b))
Π→λ-helper x = x

macro
  Π→λ : Term → Term → TC Unit.⊤
  Π→λ tm goal =  normalise tm
                 >>=_term λ tm' → checkType goal (Π→λ-type tm')
                 >>=_term λ _ →  unify goal (Π→λ-helper tm')

{- ρ :waist n  ≡  Π→λ (ρ n)  -}
macro
  _:waist_ : (pkg : Term) (height : Term) (goal : Term) → TC Unit.⊤
  _:waist_ pkg n goal = normalise (pkg app n)
                        >>=_term λ ρ → checkType goal (Π→λ-type ρ)
                        >>=_term λ _ → unify goal (Π→λ-helper ρ)

-- 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 source of a type, not an arbitrary term.
-- E.g., sources (Σ x : τ • body) = Σ x : sources_t τ • sources body
sources_t : Term → Term

{- “Π {a : A} • Ba”  ↦  𝟘 -}
sources_t (pi (arg (arg-info hidden _) A) _) = quoteTerm 𝟘

{-  “Π a : A • Π b : Ba • C a b” ↦ “Σ a : A • Σ b : B a •
↪  sources_t (C a b)”  -}
sources_t (pi (arg a A) (abs “a” (pi (arg b Ba) (abs “b” Cab)))) =
  def (quote Σ) (vArg A
                :: vArg (lam visible (abs “a”
                   (def (quote Σ)
```

```
                            (vArg Ba
                           :: vArg (lam visible (abs ‘‘b’’ (sources_t Cab)))
                           :: []))))
                  :: [])

{- ‘‘Π a : A ● Ba’’ ↦ ‘‘A’’ provided Ba does not begin with a Π  -}
sources_t (pi (arg a A) (abs ‘‘a’’ Ba)) = A

{- All other non function types have an empty source; since X ≅ (𝟙 → X) -}
sources_t _ = quoteTerm (𝟙 {ℓ_0})

{-# TERMINATING #-} -- Termination via structural smaller arguments is not clear due to the call to List.map
sources_term : Term → Term

sources_term (pi a b) = sources_t (pi a b)
{- ‘‘Σ x : τ ● Bx’’ ↦  ‘‘Σ x : sources_t τ ● sources Bx’’ -}
sources_term (def (quote Σ) (ℓ_1 :: ℓ_2 :: τ :: body))
    = def (quote Σ) (ℓ_1 :: ℓ_2 :: map-Arg sources_t τ :: List.map (map-Arg sources_term) body)

{- This function introduces 𝟙s, so let's drop any old occurances a la 𝟘. -}
sources_term (def (quote 𝟙) _) = def (quote 𝟘) []

-- TODO: Maybe we do not need these cases.
sources_term (lam v (abs s x))     = lam v (abs s (sources_term x))
sources_term (var x args) = var x (List.map (map-Arg sources_term) args)
sources_term (con c args) = con c (List.map (map-Arg sources_term) args)
sources_term (def f args) = def f (List.map (map-Arg sources_term) args)
sources_term (pat-lam cs args) =  pat-lam cs (List.map (map-Arg sources_term) args)

-- sort, lit, meta, unknown
sources_term t = t

macro
  sources : Term → Term → TC Unit.⊤
  sources tm goal = normalise tm >>=_term λ tm' → unify (sources_term tm') goal

arg-term : ∀ {ℓ} {A : Set ℓ} → (Term → A) → Arg Term → A
arg-term f (arg i x) = f x

{-# TERMINATING #-}
length_t : Term → ℕ
length_t (var x args)      = 1 + sum (List.map (arg-term length_t ) args)
length_t (con c args)      = 1 + sum (List.map (arg-term length_t ) args)
length_t (def f args)      = 1 + sum (List.map (arg-term length_t ) args)
length_t (lam v (abs s x)) = 1 + length_t x
length_t (pat-lam cs args) = 1 + sum (List.map (arg-term length_t ) args)
length_t (pi X (abs b Bx)) = 1 + length_t Bx
{-# CATCHALL #-}
```

```
-- sort, lit, meta, unknown
length_t t = 0
-- The Length of a Term:1 ends here

-- [[The Length of a Term][The Length of a Term:2]]
_ : length_t (quoteTerm (Σ x : ℕ • x ≡ x)) ≡ 10
_ = refl

--
var-dec_0 : (fuel : ℕ) → Term → Term
var-dec_0 zero t  = t
-- Let's use an “impossible” term.
var-dec_0 (suc n) (var zero args)      = def (quote 𝟘) []
var-dec_0 (suc n) (var (suc x) args)   = var x args
var-dec_0 (suc n) (con c args)         = con c (map-Args (var-dec_0 n)
↪  args)
var-dec_0 (suc n) (def f args)         = def f (map-Args (var-dec_0 n)
↪  args)
var-dec_0 (suc n) (lam v (abs s x))    = lam v (abs s (var-dec_0 n x))
var-dec_0 (suc n) (pat-lam cs args)    = pat-lam cs (map-Args (var-dec_0
↪  n) args)
var-dec_0 (suc n) (pi (arg a A) (abs b Ba)) = pi (arg a (var-dec_0 n A))
↪  (abs b (var-dec_0 n Ba))
-- var-dec_0 (suc n) (Π[ s : arg i A ] B) = Π[ s : arg i (var-dec_0 n
↪  A) ] var-dec_0 n B
{-# CATCHALL #-}
-- sort, lit, meta, unknown
var-dec_0 n t = t

var-dec : Term → Term
var-dec t = var-dec_0 (length_t t) t

{-# TERMINATING #-}
Σ→⊎_0 : Term → Term

{-  “Σ a : A • Ba” ↦ “A ⊎ B” where ‘B’ is ‘Ba’ with no reference to
↪  ‘a’  -}
Σ→⊎_0 (def (quote Σ) (h_1 ∷ h_0 ∷ arg i A ∷ arg i_1 (lam v (abs s x))
↪  ∷ []))
  =  def (quote _⊎_) (h_1 ∷ h_0 ∷ arg i A ∷ vArg (Σ→⊎_0 (var-dec x)) ∷
  ↪  [])

-- Interpret “End” in do-notation to be an empty, impossible,
↪  constructor.
-- See the unit tests above ;-)
-- For some reason, the inclusion of this caluse obscures structural
↪  termination.
Σ→⊎_0 (def (quote 𝟙) _) = def (quote 𝟘) []

 -- Walk under λ's and Π's.
Σ→⊎_0 (lam v (abs s x)) = lam v (abs s (Σ→⊎_0 x))
Σ→⊎_0 (pi A (abs a Ba)) = pi A (abs a (Σ→⊎_0 Ba))
Σ→⊎_0 t = t

macro
  Σ→⊎ : Term → Term → TC Unit.⊤
  Σ→⊎ tm goal = normalise tm >>=_term λ tm' → unify (Σ→⊎_0 tm')
  ↪  goal

{-# NO_POSITIVITY_CHECK #-}
data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
```

```
  μ : F (Fix F) → Fix F

macro
  termtype : Term → Term → TC Unit.⊤
  termtype tm goal =
              normalise tm
         >>=ₜₑᵣₘ λ tm' → unify goal (def (quote Fix) ((vArg (Σ→⊎₀ (sourcesₜₑᵣₘ tm'))) ∷ []))

-- i-th injection:  (inj₂ ∘ ⋯ ∘ inj₂) ∘ inj₁
Inj₀ : ℕ → Term → Term
Inj₀ zero c    = con (quote inj₁) (arg (arg-info visible relevant) c ∷ [])
Inj₀ (suc n) c = con (quote inj₂) (vArg (Inj₀ n c) ∷ [])

macro
  Inj : ℕ → Term → Term → TC Unit.⊤
  Inj n t goal = unify goal ((con (quote μ) []) app (Inj₀ n t))
```

A.2. Example uses of Context

These are the examples from Chapter 5, in a self-contained listing.

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

module Context_Examples where

open import Context

open import Data.Product
open import Level renaming (zero to ℓ₀; suc to ℓsuc)
open import Relation.Binary.PropositionalEquality hiding ([_])
open import Data.Empty
open import Relation.Nullary
open import Data.Nat
open import Function using (id)
open import Data.Bool renaming (Bool to 𝔹)
open import Data.Sum

open import Data.List
import Data.Unit as Unit
open import Reflection hiding (name; Type) renaming (_>>=_ to _>>=ₜₑᵣₘ_)

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

_ : Set₁
_ = DynamicSystem₀

_ : Π X : Set • Set
_ = DynamicSystem₁

_ : Π X : Set • Π x : X • Set
_ = DynamicSystem₂

idτ₀ : Set₁
idτ₀ = Π X : Set • Π e : X • X

idτ₁ : Π X : Set • Set
idτ₁ = λ (X : Set) → Π e : X • X

idτ₂ : Π X : Set • Π e : X • Set
idτ₂ = λ (X : Set) (e : X) → X

{- Surprisingly, the latter is derivable from the former -}
_ : idτ₂ ≡ Π→λ idτ₀
_ = refl

{- The relationship with idτ₁ is clarified later when we get to _:waist_ -}

DynamicSystem : Context ℓ₁
DynamicSystem = do State ← Set
                   start ← State
                   next  ← (State → State)
                   End {ℓ₀}

𝒩₀ : DynamicSystem 0   {- See the above elaborations  -}
𝒩₀ = ℕ , 0 , suc , tt

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

{- ‘Instantiaing’ State to be ℕ in “DynamicSystem 1” -}

𝒩₁′ : let State = ℕ in Σ start : State  • Σ s : (State → State)  • 𝟙 {ℓ₀}
𝒩₁′ = 0 , suc , tt

_ = Π→λ (DynamicSystem 2)
  ≡⟨ "Definition of DynamicSystem at exposure level 2" ⟩'
    Π→λ (Π X : Set • Π s : X • Σ n : (X → X)  • 𝟙 {ℓ₀})
  ≡⟨ "Definition of Π→λ; replace a ‘Π’ by a ‘λ’" ⟩'
    (λ (X : Set) → Π→λ (Π s : X • Σ n : (X → X)  • 𝟙 {ℓ₀}))
  ≡⟨ "Definition of Π→λ; replace a ‘Π’ by a ‘λ’" ⟩'
    (λ (X : Set) → λ (s : X) → Π→λ (Σ n : (X → X)  • 𝟙 {ℓ₀}))
  ≡⟨ "Next symbol is not a ‘Π’, so Π→λ stops" ⟩'
    λ (X : Set) → λ (s : X) → Σ n : (X → X)  • 𝟙 {ℓ₀}

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 = ⟨⟩

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}

D₁ = DynamicSystem 0

1-records : D₁ ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • 𝟙 {ℓ₀})
1-records = refl

D₂ = DynamicSystem :waist 1

2-funcs : D₂ ≡ (λ (X : Set) → Σ z : X • Σ s : (X → X) • 𝟙 {ℓ₀})
2-funcs = refl

_ : sources (𝔹 → ℕ) ≡ 𝔹
_ = refl

_ : sources (Σ f : (ℕ → 𝔹) • Set) ≡ (Σ x : ℕ • Set)
_ = refl

_ : sources (Σ f : (ℕ → Set → 𝔹 → ℕ) • 1 ≡ 1) ≡ (Σ x : (ℕ ×
↪ Set × 𝔹) • 1 ≡ 1)
_ = refl

_ : ∀ {ℓ} → sources (𝟙 {ℓ}) ≡ 𝟘
_ = refl

_ = (sources (∀ {x : ℕ} → ℕ)) ≡ 𝟘
_ = refl {ℓ₁} {Set} {𝟘}

D₃ = sources D₂

3-sources : D₃ ≡ λ (X : Set) → Σ z : 𝟙 • Σ s : X • 𝟘
3-sources = refl

_ : Σ→⊎ (Π S : Set • (S → S)) ≡ (Π S : Set • (S → S))
_ = refl

_ : Σ→⊎ (Π S : Set • Σ n : S • S) ≡ (Π S : Set • S ⊎ S)
_ = refl

_ : Σ→⊎ (λ (S : Set) → Σ n : S • S) ≡ λ S → S ⊎ S
_ = refl

_ : Σ→⊎ (Π S : Set • Σ s : S • Σ n : (S → S) • 𝟙 {ℓ₀}) ≡ (Π S :
↪ Set • S ⊎ (S → S) ⊎ 𝟘)
_ = refl
```

```
_ : Σ→⊎ (λ (S : Set) → Σ s : S • Σ n : (S → S) • 𝟙 {ℓ₀}) ≡ λ S
↪  → S ⊎ (S → S) ⊎ 𝟘
_ = refl

D₄ = Σ→⊎ D₃

4-unions : D₄ ≡ λ X → 𝟙 ⊎ X ⊎ 𝟘
4-unions = refl

module free-dynamical-system where

    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

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

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

module termtype[Monoid]≅TreeSkeleton where

  𝕄 : Set
  𝕄 = termtype (Monoid ℓ₀ :waist 1)

  that-is : 𝕄 ≡ Fix (λ X → X × X × 𝟙 -- _⊕_, branch
                          ⊎ 𝟙         -- Id, nil leaf
                          ⊎ 𝟘         -- invariant leftId
                          ⊎ 𝟘         -- invariant rightId
                          ⊎ 𝟘         -- invariant assoc
                          ⊎ 𝟘)        --  the “End {ℓ}”
  that-is = refl

  -- 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

  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)

  from∘to : ∀ m → from (to m) ≡ m
  from∘to emptyM        = refl
```

```
  from∘to (branchM l r) = cong₂ branchM (from∘to l) (from∘to r)
  from∘to (absurdM (inj₁ ()))
  from∘to (absurdM (inj₂ ()))

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

module termtype[Collection]≅List where

  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))

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

  from : ∀ {E} → List E → ℂ E
  from []       = ∅
  from (x ∷ xs) = x :: from xs

  to∘from : ∀ {E} (xs : List E) → to (from xs) ≡ xs
  to∘from []       = refl
  to∘from (x ∷ xs) = cong (x ∷_) (to∘from xs)

  from∘to : ∀ {E} (e : ℂ E) → from (to e) ≡ e
  from∘to (e :: es) = cong (e ::_) (from∘to es)
  from∘to ∅         = refl
```