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

July 10, 2020

PhD Thesis

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

Abstract

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

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

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

Contents

Chapter 1

Introduction

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

The framework developed in this thesis is motivated by the following concerns when developing libraries in the dependently-typed language (DTL) Agda, such as [Kah18].

1. **Practical$_1$: Renaming** There is excessive repetition in the simplest of tasks when working with packages; e.g., to *uniformly* decorate the names in a package with subscripts $_0$, $_1$, $_2$ requires the package's contents be listed thrice. It would be more economical to *apply* a renaming *function* to a package.

2. **Practical$_2$: 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*. Unfortunately, library developers generally provide only a few *variations* on *a* package; such as having no parameters or having only *functional symbols* as parameters —c.f., the carrier `C` and operation $\oplus$ in figure 1. Whereas functions can *bundle-up* or *unbundle* their parameters using currying and

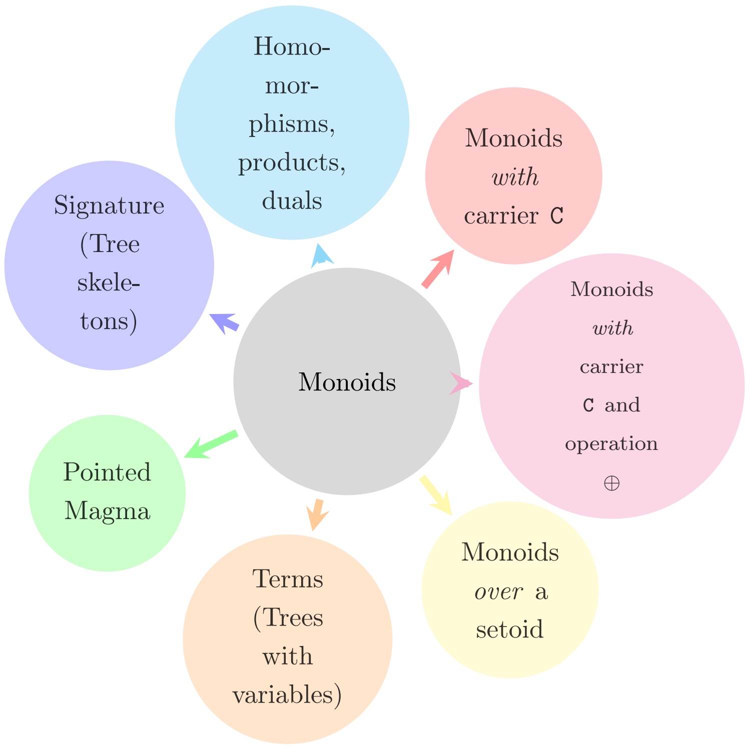


Figure 1.1: Deriving related *types* from *the* definition of monoids

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.

3. **Theoretical$_1$: Exceptionality** DTLs blur the distinguish between expressions and types, treating them as the same thing: *Terms*. This collapses a number of seemingly different language constructs into the same thing —e.g., programs and proofs are essentially the same thing. Unfortunately, packages are treated as *exceptional* values that differ from *usual* values —such as functions and numbers— in that the former are ‘second-class citizens’ which only serve to collect the latter ‘first-class citizens’. This forces users to learn two families of ‘sub-languages’ —one for each citizen class. There is essentially no *theoretical* reason why packages do not deserve first-class citizenship, and so receive the same treatment as other *unexceptional* values. Another advantage of giving packages equal treatment is that we are inexorably led to wonder what **computable algebraic structure** they have and how they relate to other constructs in a language; e.g., packages are essentially record-valued functions.

4. **Theoretical$_2$: Syntax** It is well known that sequences of declarations may be grouped together within a *package*. If any declarations are opaque, not fully undefined, they become *parameters* of the package —which may then be identified as a *record type* with the opaque declarations called *fields*. However, when a declaration is *intentionally*

opaque not because it is missing an implementation, but rather it acts as a value construction itself then one uses *algebraic data types*, or ‘termtypes’. Such types share the general structure of a package, and so it would be interesting to illuminate the exact difference between the concepts —*if any*. In practice, one forms a record type to model an interface, instances of which are actual implementations, and forms an *associated* termtype to *describe computations* over that record type, thereby making available a syntactic treatment of the interface —textual substitution, simplification / optimisation, evaluators, canonical forms. For example, as shown in figure 1, the record type of monoids models composition whereas the (tremendously useful) termtype of binary trees acts as a description language for monoids. The *problem of maintenance* now arises: Whenever the record type is altered, one must mechanically update the associated termtype. It would be more economical to extract *both* record types and termtypes from a single package declaration.

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

1.1 Thesis Overview

The remainder of the thesis is organised as follows.

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

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

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

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

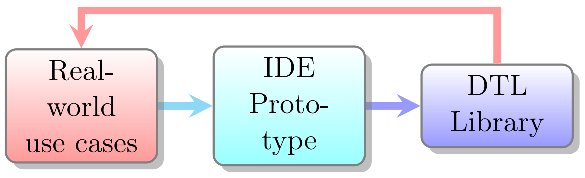


Figure 1.2: Approach for a **practical** framework

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

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

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

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

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

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

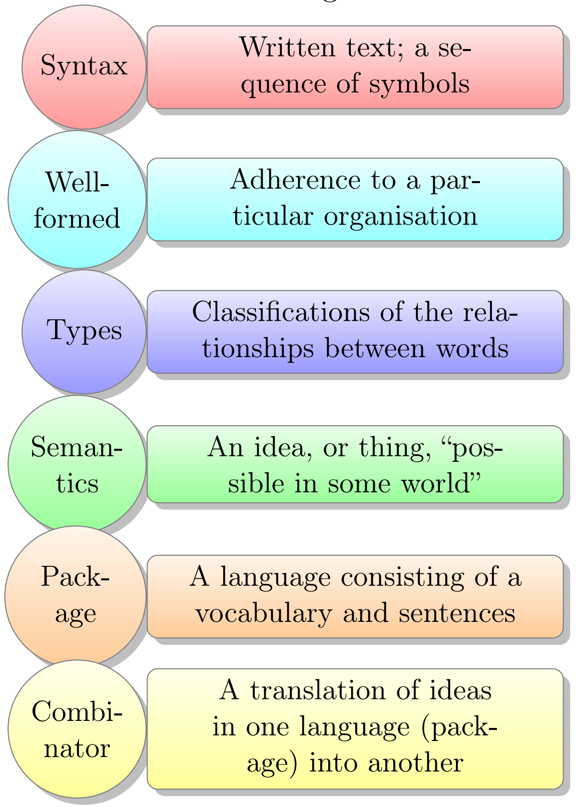
◇ Chapter 6 concludes with a discussion about the results presented in the thesis.

The underlying motivation for the research is the conviction that packages play *the* crucial role for forming compound computations, subsuming *both* record types and termtypes. The approach followed is summarised in figure 1.1.

Chapter 2

Packages and Their Parts

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


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

Game-Play Analogy

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

Programming, as is the case with all of mathematics, is the manipulation of symbols according to specific *rules*. Moreover, like a game, when one plays —i.e., shuffles symbols around— one may interpret the game pieces and the actions to *denote* some meaning, such as reflecting aspects of the players or of reality. Many play because it is fun to do so; there are only pieces (mathematical symbols or *terms*) and rules to be followed, and nothing more. Complex games may involve a number of pieces (terms) which are classified by the *types* of roles they serve, and the rules of play allow us to make observations or *judgements* about them; such as, “in the stage 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 rules of the game are its *syntax* and what the game means is its *semantics*. To say that a game piece (term) denotes some idea **I**, we need to be able to *express* that idea which may only be possible in the meta-language; e.g., pieces in a mini-game within a game may themselves denote pieces within the primary game —more concretely, a game may require a roll of a die whose numbers *denote*, or *refer to*, players in the main game which are not expressible in the mini-game. A *model* of a game (type theory) is an interpretation of the game’s pieces in way that the rules are true under the interpretation.

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

Machine

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

Colours

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

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

Subsection Goals

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

The goals of the subsections are as follows.

Provide a formalism of the above `Colour` package

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

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

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

Demonstrate the interdefinability of structuring mechanisms

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

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

Take inspiration from how other DTLs handle packages

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

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

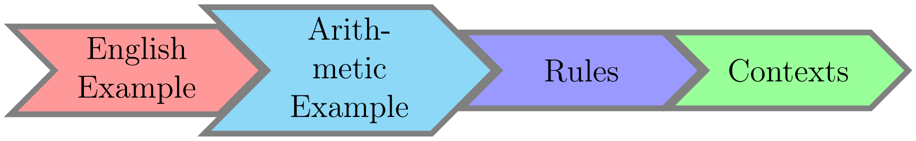
Contributions of the thesis

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

9. What are the outcomes of the thesis effort.

2.1 What is a language?

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



Madlips[1]: Simple English sentences have the form subject-verb-object such as *"Jim ate the apple"*. To *mindlessly* produce such sentences, one must produce a subject, then a verb, then an object —all from given lists of possibilities. A convenient notation to describe a language is its *grammar* [Cho59a; Cho59b] presented in *Backus-Naur Form* [CCH73; GDF02; Lar+11; Knu64] as in Figure 2.1.

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

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

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

Figure 2.1: Madlips Grammar

Example Derivation

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

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

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

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

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

Freshmen Grammar

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

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

Example terms include the numbers `Zero`, `Succ Zero`, and `Succ Succ Zero` —which denote 0, 1 (the successor of zero), and 2 (the successor of the successor of zero). The sensible Booleans terms `True ≈ False` and `True` are also possible —regardless of *how true*

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

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 : τ` is defined by (1) introducing a new syntactic category (called "types") to 'tag' terms with the kind of elements they denote, and (2) declaring the conditions under which the relationship is true.

Types for Freshmen

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

Judgement Rules

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

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

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

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

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

Concept	Intended Interpretation
type	a collection of things
term	a particular one of those things
$x : \tau$	the declaration that x is indeed within collection 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 now must now additionally keep track of the context; e.g., the `_+_` rule becomes:

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

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

Before we move on, it is interesting to note that contexts can themselves be presented with a grammar —as shown below, where constructors ‘,’ and ‘:’ each take two arguments and are written infix; i.e., instead of the usual `, arg₁ arg₁` we write `arg₁ , arg₂`. Contexts are *well-formed* when variables are associated at most one type; i.e., when contexts *represent* ‘partial functions’.

Grammar for Contexts

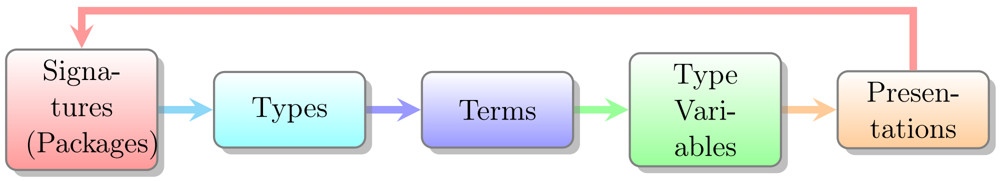
```
Context     ::= ∅ | Association, Context
Association ::= Variable : Type
```

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

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

2.2 Signatures

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



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

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

◇ a set $\mathcal{F}$ of *function symbols*, and

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

Signatures generalise graphical sketches

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

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

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

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

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

Grammar for Arbitrary Terms

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

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

Figure 2.2: Signature Typing

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

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

[3] We write `List X` for the type of lists with values from `X`. The empty list is written `[]` and `[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.

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

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

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

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

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

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

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

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

Freshmen as a Generalised Signature

```
Number  : Type
Boolean : Type

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

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

What a twist: **Generalised signatures are contexts!** That is, a sequence of name-type associations. More precisely, with the relation `package_` defined below, we can characterise

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

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

packages as the contexts whose earlier elements allow their later elements to be typeable. For example, the context `S : Type; x : S` can be proven to be package whereas the context `S : Type; x : Q` cannot —it has the 'global name' Q.

Rules for determining when a signature is a package

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

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

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

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

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

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

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

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

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

Table 2.1: Three "typing" relations

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

Induced Grammar for Types

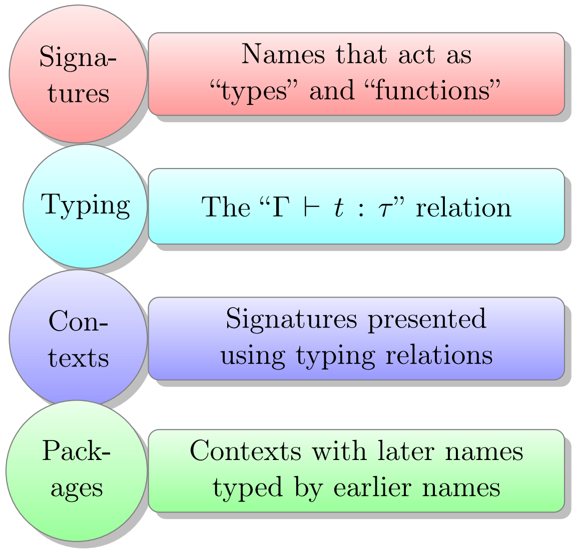
```
TYPE ::= Type                -- An opaque symbol; “the type of types”
      |  τ                   -- τ is a sort symbol; a value of 𝒮
      |  x                   -- A variable; an element of 𝒱
      |  TYPE → TYPE          -- _→_ takes two TYPE arguments
      |  TYPE × TYPE |  𝟙 -- “product types”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

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

We may now form types $\alpha \to \beta$ 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 `s ≈ t` to be a well-formed term of type `Boolean` *provided* `s` and `t` have the *same* type, either `Number` or `Boolean`. That is, `_≈_` wants to have *both* `Number × Number → Boolean` *and* `Boolean × Boolean → Boolean` as types —since it is reasonable to compare either numbers *or* truth values for equality. But a function symbol can have only *one* type —since `src` and `tgt` are (deterministic) functions. If we had access to variables which stand-in for types, we could type equality as `α × α → Boolean` *for any type* α.

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

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, 𝟙, _×_, _→_` *are* function symbols, then `TYPE` is subsumed by `Term`. Hence, we may conflate the two into one declaration to obtain *dependently-typed terms* —a concern which we will return to at a later time. For now, we may summarise our progress with the following figure.


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

2.3 Presentations of Signatures —Π and Σ

Since a signature's types also have a grammar, we can present a signature in the natural style of "name : type-term" pairs. That is, a signature may be presented as a context; i.e., sequence of declarations δ_0, δ_1, ..., 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. For example, the above presentation of Freshmen is a context from which we regain a signature $\Sigma = (\mathcal{S}, \mathcal{F}, src, tgt)$ where:

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

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

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

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

Calling-smart-people Context

```
Location : Type

School   : Location
Street   : Location
TV       : Location

Smart    : Location → Type

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

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

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

Dependent Function Type

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

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

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

Before extending the previous definition of signatures, there is a practical subtlety to consider. Suppose we want to talk about smart people *regardless* of their location, how would you express such a type? The type of `call` : ($\Pi\ l$: `Location` $\bullet$ `Smart` $l \to$ `Phone`) reads: *After picking a particular location ℓ, you may get the phone numbers of the smart people at that location.* More specifically, $\Pi\ \ell$: `Location` $\bullet$ `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 as a particular value, is known as the **(un)bundling problem**[8]. It is addressed by introducing a new[9] type operator $\Sigma\_{:}\_ \bullet \_$ —the symbol ‘Σ’ is conventionally used both for the name of signatures and for this new type operator.

[7]Those familiar with set theory may remark that dependent types are not *necessary* in the presence of power sets: Instead of a *single* name `call`, one uses a (possibly infinite) *family of names* 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 $\forall/\exists$ encoded via dependent types Π/Σ; whence the slogan “Programming $\approx$ Proving”.

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

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

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

Dependent Product Type

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

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

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

Abbreviations

Provided B is a type that does not vary,

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

Since 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.

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

Example 1: People and their birthdays

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

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

Example functions in this type are f and g below…

```
f Monday  = Jim
f Tuesday = Alice

g Monday  = Mark
g Tuesday = Alice
```

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

Person	Birthday
Jim	Monday
Alice	Tuesday
Mark	Monday

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

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

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

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

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

Example 2: English words and their lengths

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

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

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

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

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

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

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

In contrast, $\mathbb{N} \times$ `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!**

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

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

Generalised Terms

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

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

Intended Interpretations of Generalised Terms

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

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

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

The rules below classify the well-formed generalised terms. The rules for 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 upcoming definition.

Judgements for Generalised Terms

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

The Variables rule is also known as Assumption and may be rendered as follows.

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

. . . .

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

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

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

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

. . . .

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

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

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

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

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.

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

Judgement for Generalised Signatures

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

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

Pointed Families

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

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

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

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

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

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

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

Signatures are a staple of computing science since they formalise interfaces and generalise graphs and type theories. Our generalised signatures have been formalised "after the fact" from the creation of the prototype for packages. In the literature, our definition of generalised signatures is essentially a streamlined presentation of Cartmell's *Generalised Algebraic Theories* [Car86] expect that we do not allow arbitrary equational 'axioms' instead using "name = term" rather than "term = term" axioms which serve as *default implementations* of names. We now turn to extending the current setup to permit optional definitions.

Permitting Optional Definitions

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

A dynamical system – Colours

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

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

A **Generalised Signature** is now defined to be a tuple *($\mathcal{B}$, type, definition)* where $\mathcal{B}$ = *[β_0, β_1, ..., β_n]* is an *ordered* list of "base symbols", `type` : $\mathcal{B} \to$ `Term` associates a generalised term to each base symbol such that $\Gamma_{k-1} \vdash \tau_k$: `Type` for each k : *0..n*, where $\Gamma_k = (\beta_0 : \tau_0, \ldots, \beta_k : \tau_k)$ and $\tau_i =$ `type` β_i; and `definition` : $\mathcal{B} \to$ `Term` is a partial function associating a term to each symbol name such that the types agree: $\Gamma_{k-1} \vdash$ `definition` β_k : `type` β_k. That is `type` associates to each base symbol a type-term that is well-defined according to the typing rules above for generalised terms and *possibly* making use of previous symbols in the listing. Then `definition` β_k *may* provide a description of a value of `type` β_k.

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

Example 4: Colours as Generalised Signatures

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

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

Example 5: Disjoin Sums as Generalised Signatures

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

Sums from Σ and 𝔹

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

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

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

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

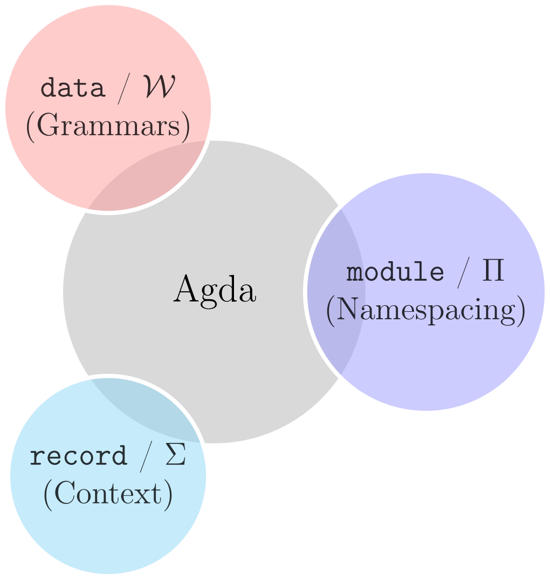
Of course contexts now associate *both* a type and an optional definition with a given name, and so $\Gamma : \mathsf{Name} \to \mathsf{Term} \times (\mathsf{Term} \cup \{-\})$ where "$-$" denotes "no definition". That is, we essentially have two judgement relations $\Gamma \vdash \eta : \tau$ and $\Gamma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

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

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

2.4 A Whirlwind Tour of Agda

We have introduced a number of concepts and it can be difficult to keep track of when relationships $\Gamma \vdash t : \tau$ are in-fact derivable. The Agda McKinna [McK06], McBride [McB00], Bove and Dybjer [BD08], and Wadler and Kokke [WK18] 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`.



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 $\mathcal{N}$, $_7$, ::, $\to$. Moreover, the Emacs interface allows programming by gradual refinement of incomplete type-correct terms. One uses the "hole" marker `?` as a placeholder that is used to stepwise write a program.

2.4.1 Dependent Functions — Π-types

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

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

The Identity Function

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

id₁ id₂ id₃ : (X : Set) → X → X

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

All these functions explicitly require the type `X` when we use them, which is silly since it can be inferred from the element `x`. Curly braces make an argument *implicitly inferred* and so it may be omitted. E.g., the `{X : Set} → ⋯` below lets us make a polymorphic function since `X` can be inferred by inspecting the given arguments. This is akin to informally writing 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 x₁ x₂ … xₙ`. The `infix` declaration indicates how complex expressions involving the new notation should be parsed; in this case, the new notation binds less than any operator in Agda.

2.4.2 Dependent Datatypes — ADTs

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

Informal Grammar Notation

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

Agda Rendition of Grammars

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

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

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

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

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

Interpreting the Door Values as Options

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

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

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

`Arabic` 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.

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

Table 2.2: Many useful ideas arise as grammars

Vectors —ℕ-indexed Lists

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

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

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

Safe Head

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

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

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

Catenation is a $++ \longrightarrow +$ Homomorphism

```
infixr 40 _++_

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

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

2.4.3 ADT Example: Propositional Equality

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

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

Propositional Equality

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

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

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

Transport along proofs

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

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

Propositional equality is the least reflexive relagtion

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

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

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

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

Examples of Propositional and Definitional Equality

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

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

2.4.4 ADTs as $\mathcal{W}$-types

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

Describing Possibly Non-Existent Languages

```
data N : Set where
  MakeN : N

data No : Set where
  MakeNo : No → No

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

How do we know if a grammar describes a language that *actually exists*? Suppose `T` is defined by n constructors $\mathtt{C}_i : \tau_i(\mathtt{T}) \to \mathtt{T}$, 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 `N` above has one constructor `MakeN` which takes an empty tuple of arguments, denoted $\mathbb{1} = \{\ ()\ \}$, and so it has $\mathbf{F}\ \mathtt{X} \approx \mathbb{1}$ and so $(\mathbf{F}\ \mathtt{X} \to \mathtt{X}) \approx (\mathbb{1} \to \mathtt{X}) \approx \mathtt{X}$; 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 `MakeN`. **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: $\mathcal{W}$ `A B` has `A` as 'function symbols' and each symbol `f : A` has 'type' `B f`. $\mathcal{W}$-types are not generalised signatures since they do not support optional definitions; which is a minor technicality: If t has the associated definition `d`, then we may use " `let t = d in` $\mathcal{W}$ $\cdots$ " and repeated `let` clauses solve the issue of optional definitions.

The generic situation of 'containers' is described in [Alt+15].

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"[11] descriptions $\mathbf{F}\ \mathtt{X} = (\Sigma\ \mathtt{a} : \mathtt{A} \bullet \Pi\ \mathtt{b} : \mathtt{B\ a} \bullet \mathtt{X})$ actually describe a language. That is, when is there

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

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

Descriptions of Languages That Necessarily Exist

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

$\mathcal{W}$-types in Agda

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

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

[a]See [Uni13] and [Emm18] for an introduction to $\mathcal{W}$-types. The dual, non-well-founded (coinductive) types, are called *M-types* and they are derivable from W-types; see [ACS15]. The generic concept of 'containers' is described in [Alt+15].

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

[12]Categorically speaking, polynomial functors —i.e., type formers of the shape `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 languages described by $\mathbf{F}$. That is, $\mathcal{W}$-types are the initial algebras of polynomial functors; that is, $\mathbf{F}$ has an initial algebra `sup :` $\mathbf{F}\ \mathbf{W} \to \mathbf{W}$. Moreover, every strictly positive type operator can be expressed in the same shape as $\mathbf{F}$ and so they all have an initial algebra —for details see [Dyb97]. Inductive families arise as indexed $\mathcal{W}$-types which are initial algebras for dependent polynomial functors, and [GH04] 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 [AAG04].

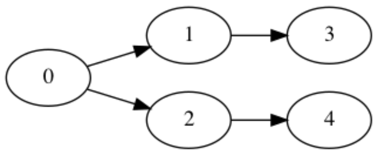
[13]Since every functon `Fin n → X` can be 'tabulated' as a `List X` value of length `n`
—i.e., `(Σ xs : List A • length xs ≡ n) ≅ (Fin n → A)`— we have that `Rose' A ≅ Rose A`.

Rose trees

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

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

The `example` tree is shown diagrammatically below.



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

Rose trees

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

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

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

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

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

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

$$\sup \emptyset = 0$$

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

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

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

2.4.5 Modules —Namespace Management; ΠΣ-types

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

A Simple Module

```
module A where

  𝒩 : Set
  𝒩 = ℕ

  private
    x : ℕ
    x = 3

  y : 𝒩
  y = x + 1
```

Using It

```
use₀ : A.𝒩
use₀ = A.y

use₁ : ℕ
use₁ = y
  where open A

open A

use₂ : ℕ
use₂ = y
```

Parameterised Modules

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

Name = Function

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

Using Them

```
use'₀ : ℕ
use'₀ = B.y 3

module C = B 3

use'' : ℕ
use'' = C.y

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

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

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

[15] A 𝒲-type is empty precisely when it has no nullary constructor; See exercise 5.17 of [Uni13].

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

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

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

Table 2.3: Module combinators supported in the current implementation of Agda

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

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

2.4.6 Records — Σ-types

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

$$\texttt{record} \approx \texttt{module} + \texttt{data}\ \text{with one constructor}$$

The class of types along with a value picked out

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

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

Defining Instances

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

ex₁ : PointedSet
ex₁ = MkIt ℕ 3

open PointedSet

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

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

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

Simple Uses

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

use¹ : ℕ
use¹ = point
    where open PointedSet ex₀

open PointedSet

use² : ℕ
use² = blind ex₀ true
```

Pattern Matching on Records

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

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

use⁴ (MkIt C x)
  = x
```

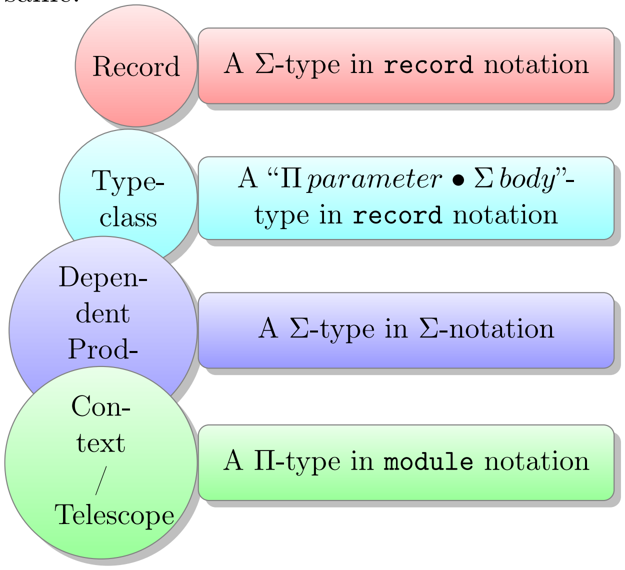
Records are `data` declarations whose one and only constructor is named `record {f`$_i$` = _}`, where the `f`$_i$ are the filed names; above we provided `MkIt` as an optional alias. As such, above we could pattern match on records using either constructor name.

So much for records.

2.5 Facets of Structuring Mechanisms

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

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


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

2.5.1 Three Ways to Define Monoids

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

Monoids as Agda Records

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

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

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

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

Instance Resolution

The double curly-braces `{{...}}` serve to indicate that the given argument is to be found by *instance resolution*. For example, if we declare `it : {{e : A}} → B`, then `it` is a `B` value that is formed using an `A` value; but which `A` value? Unlike a function which requires the `A` value as input, `it` will "look up" an `A` value in the list of names that are marked for look-up by the keyword `instance`. If multiple `A` values are marked for look-up, it is not clear which one should be used; as such, *at most one*[a] value can be provided for lookup and this value is called "the declared `A`-instance", whence the name 'instance resolution'. Recall that Agda records automatically come with an associated module, and so the `open` clause, above, makes the name `pop-Id-Rec : {{M : Monoid-Record}} → (x y : Monoid-Record.Carrier M) → ...` accessible; in-particular, this name uses instance resolution: The derived result, `pop-Id-Rec`, can be invoked without having to mention a monoid, provided a unique `Monoid-Record` value is declared for instance search —otherwise one must use named instances Kahl and Scheffczyk [KS01]. We will return to actually declaring and using instances in the next section.

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

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

Monoids as Typeclasses

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

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

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

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

Monoids as Dependent Sums

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

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

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

e.g.,

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

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

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

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

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

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

2.5.2 Instances and Their Use

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

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

Instance Declarations

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

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

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

Interestingly, notice that the grouping in ℕ-Σ is just an unlabelled (dependent) product, and so when it is used below in `pop-Id-Σ` we project to the desired components. Whereas in the `Monoid-Record` case we could have projected the carrier by `Carrier M`, now we would write $\texttt{proj}_1$ `M`.

No Monoids Mentioned at Use Sites

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

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

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

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

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

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

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

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

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

2.5.3 A Fourth Definition —Contexts

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

Monoids as Telescopes

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

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

Squint and They're The Same

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

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

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

Agda

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

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

Agda

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

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

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

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

Agda

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

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

Agda

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

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

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

2.6 Contexts are Promising

The current implementation of the Agda language Bove, Dybjer, and Norell [BDN09] and Norell [Nor07] 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 Pientka [Pie10]. A term `t(x)` may have free variables and so whether it is well-formed, or what its type could be, depends on the types of its free variables, necessitating one to either declare them before hand or to write, in Beluga,

`[ x : T |- t(x) ]` for example. As argued in the previous section, contexts are essentially dependent sums. In contrast to Beluga, **Isabelle** is a full-featured language and logical framework that also provides support for named contexts in the form of 'locales' Ballarin [Bal03] and Kammüller, Wenzel, and Paulson [KWP99]; unfortunately it is not a dependently-typed language.

◇ **Signatures as an underlying formalism**

Twelf Pfenning and Team [PT15] is a logic programming language implementing Edinburgh's Logical Framework Urban, Cheney, and Berghofer [UCB08], Rabe [Rab10], and Stump and Dill [SD02] and has been used to prove safety properties of 'real languages' such as SML. A notable practical module system Rabe and Schürmann [RS09] for Twelf has been implemented using signatures and signature morphisms.

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

The current implementation of **Coq** Paulin-Mohring [Pau] and Gross, Chlipala, and Spivak [GCS14] provides a "copy and paste" operation for modules using the `include` keyword. Consequently it provides a number of module combinators, such as `<+` which is the infix form of module inclusion Coq Development Team [Coq18]. Since Coq module types are essentially contexts, the module type `X <+ Y <+ Z` is really the catenation of contexts, where later items may depend on former items. The **Maude** Clavel et al. [Cla+07] and Durán and Meseguer [DM07] framework contains a similar yet more comprehensive algebra of modules and how they work with Maude theories.

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

2.7 Coq Modules as Generalised Signatures

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

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

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

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

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

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

Graphs —Coq

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

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

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

Graphs —Agda

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

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

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

Birds' Eye View

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

Booleans are Graphs —Coq

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

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

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

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

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

Booleans are Graphs —Agda

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

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

Minimisation as a function on modules —Coq

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

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

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

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

Minimisation as a function on modules —Agda

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

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

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

open Min
```

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

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

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

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

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

```
Conjunction = Min BoolGraph

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

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

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

A module-to-module function —Coq

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

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

A module-to-module function —Agda

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

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

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

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

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

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

  max : Vertex → Vertex → Vertex
  max = min I

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

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

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

Table 2.4: 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.

Table 2.5: Modules and module types only differ in intended utility

2.8 Problem Statement, Objectives, and Methodology

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

2.8.1 Problem Statement

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

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

This is understandable: Function symbols generally vary more often than proof obligations. (This is discussed in detail in Section **??**.) 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 [Alt+15]; in which case, they will likely have to create it by hand.

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

2.8.2 Objectives and Methodology

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

◇ **Objective 1: Modelling Module Systems**

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

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

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

◇ **Objective 2: Support Unexpected Notions of Module**

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

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

same' and others as 'different' to obtain a new module that has copies of some parameters but not others. Moreover, users should be able to mechanically produce the necessary morphisms to make this construction into a pushout. Likewise, we would expect products, unions, intersections, and substructures of theories —when possible, and then to be constructed by users. In this thesis, we only want to provide a fixed set of meta-primitives from which usual and (un)conventional notions of grouping may be defined.

◇ **Objective 3: Provide a Semantics**

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

2.9 Contributions

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

1. The ability to model module systems *for* DTLs *within* DTLs
2. The ability to arbitrarily *extend* such systems by users at a high-level
3. Demonstrate that there is an expressive yet minimal set of module meta-primitives which allow common module constructions to be defined
4. Demonstrate that relationships between modules can also be *mechanically* generated.
   - ◇ In particular, if module $\mathcal{B}$ is obtained by applying a user-defined 'variational' to module $\mathcal{A}$, then the user could also enrich the child module $\mathcal{B}$ with morphisms that describe its relationships to the parent module $\mathcal{A}$.
   - ◇ E.g., if $\mathcal{B}$ is an extension of $\mathcal{A}$, then we may have a "forgetful mapping" that drops the new components; or if $\mathcal{B}$ is a 'minimal' rendition of the theory $\mathcal{A}$, then we have a "smart constructor" that forms the rich $\mathcal{A}$ by only asking the few $\mathcal{B}$ components of the user.
5. Demonstrate that there is a *practical* implementation of such a framework
6. Solve the unbundling problem: The ability to 'unbundle' module fields as if they were parameters 'on the fly'
   - ◇ I.e., to transform a type of the shape $\Pi^w\Sigma$ into $\Pi^{w+k}\Sigma$, for $k \geq 0$, such that the resulting type is *as practical and as usable* as the original

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

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

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

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

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

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

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

Bibliography

[AAG04] 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 Dı′az 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 (cit. on p. 41).

[ACS15] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. “Non-wellfounded trees in Homotopy Type Theory”. In: *CoRR* abs/1504.02949 (2015). arXiv: 1504.02949. URL: http://arxiv.org/abs/1504.02949 (cit. on p. 41).

[Alt+15] Thorsten Altenkirch et al. “Indexed containers”. In: *J. Funct. Program.* 25 (2015). DOI: 10.1017/S095679681500009X. URL: https://doi.org/10.1017/S095679681500009X (cit. on pp. 40, 41, 61).

[Ast+02] Egidio Astesiano et al. “CASL: the Common Algebraic Specification Language”. In: *Theor. Comput. Sci.* 286.2 (2002), pp. 153–196. DOI: 10.1016/S0304-3975(01)00368-1. URL: https://doi.org/10.1016/S0304-3975(01)00368-1 (cit. on p. 49).

[Bal03] 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 (cit. on p. 54).

[BD08] Ana Bove and Peter Dybjer. “Dependent Types at Work”. In: *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*. 2008, pp. 57–99. DOI: 10.1007/978-3-642-03153-3_2. URL: https://doi.org/10.1007/978-3-642-03153-3%5C_2 (cit. on p. 33).

[BDN09] 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 (cit. on p. 53).

[Car86] 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` (cit. on p. 30).

[CCH73] 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` (cit. on p. 12).

[Cho59a] 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` (cit. on p. 12).

[Cho59b] 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` (cit. on p. 12).

[Cla+07] 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` (cit. on p. 54).

[Coq18] 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` (cit. on p. 54).

[DM07] 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` (cit. on p. 54).

[Dyb97] 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` (cit. on p. 41).

[Emm18] Jacopo Emmenegger. *W-types in setoids*. 2018. arXiv: `1809.02375v2 [math.LO]` (cit. on p. 41).

[GCS14] Jason Gross, Adam Chlipala, and David I. Spivak. *Experience Implementing a Performant Category-Theory Library in Coq*. 2014. arXiv: `1401.7694v2 [math.CT]` (cit. on p. 54).

[GDF02] 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 (cit. on p. 12).

[GH04] 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 (cit. on p. 41).

[Kah18] Wolfram Kahl. *Relation-Algebraic Theories in Agda*. 2018. URL: http://relmics.mcmaster.ca/RATH-Agda/ (visited on 10/12/2018) (cit. on p. 3).

[Knu64] 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 (cit. on p. 12).

[KS01] Wolfram Kahl and Jan Scheffczyk. "Named Instances for Haskell Type Classes". In: 2001 (cit. on p. 47).

[KWP99] 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 (cit. on p. 54).

[Lar+11] 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 (cit. on p. 12).

[McB00] Conor McBride. "Dependently typed functional programs and their proofs". PhD thesis. University of Edinburgh, UK, 2000. URL: http://hdl.handle.net/1842/374 (cit. on p. 33).

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

[Nor07] 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 (cit. on p. 53).

[Pau] Christine Paulin-Mohring. "The Calculus of Inductive Definitions and its Implementation: the Coq Proof Assistant". In: invited tutorial (cit. on p. 54).

[Pie10] 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 (cit. on p. 53).

[PT15] Frank Pfenning and The Twelf Team. *The Twelf Project*. 2015. URL: http://twelf.org/wiki/Main_Page (visited on 10/19/2018) (cit. on p. 54).

[Rab10] 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 (cit. on p. 54).

[RS09] Florian Rabe and Carsten Schürmann. “A practical module system for LF”. In: *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP '09, McGill University, Montreal, Canada, August 2, 2009*. 2009, pp. 40–48. DOI: 10.1145/1577824.1577831. URL: https://doi.org/10.1145/1577824.1577831 (cit. on p. 54).

[SD02] 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 (cit. on p. 54).

[UCB08] Christian Urban, James Cheney, and Stefan Berghofer. *Mechanizing the Metatheory of LF*. 2008. arXiv: 0804.1667v3 [cs.LO] (cit. on p. 54).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cit. on pp. 41, 43).

[WK18] Philip Wadler and Wen Kokke. *Programming Language Foundations in Agda*. 2018. URL: https://plfa.github.io/ (visited on 10/12/2018) (cit. on p. 33).