The Next 700 Module Systems

Extending Dependently-Typed Languages to Implement Module System Features In The Core Language

Department of Computing and Software

McMaster University

Musa Al-hassy

May 8, 2019

Thesis Proposal

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

Abstract

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

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

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

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

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

Contents

Chapter 1

Introduction —The Proposal's "Story"

In this chapter we aim to present the narrative that demonstrates the distinction between what can currently be accomplished and what is desired when working with composition of software units. We arrive at the observation that packaging concepts differ only in their use –for example, a typeclass and a record are both sequences of declarations that only differ in the former used for polymorphism with instance search whereas the latter is used as a structure grouping related items together. In turn, we are led to propose that the various packaging concepts ought to have a uniform syntax. Moreover, since records are a particular notion of packaging, the commitment to syntactic similarity gives rise to a homoiconic nature to the host language.

Within this work we refer to a *simple type theory* as a language that contains typed lambda terms for terms and formuale; if in addition it contains typed lambda terms for 'proofs' —which are members of types that could be interpreted as propositions— then we say it is a *dependently-typed language*, or 'DTL' for short. More precisely, if type formation is indexed, i.e., types may depend on a context, then we have a DTL. With the exception of declarations and ephemeral notions, nearly everything in a DTL is a typed lambda term. Just as Lisp's homoiconic nature blurs data and code leaving it not as a language with primitives but rather a language with meta-primitives, so too the lack of distinction between term and type lends itself to generic and uniform concepts in DTLs thereby leaving no syntactic distinction between a constructive proof and an algorithm.

The sections below explore our primary observation, which is discussed further later on in chapter 3 as preliminary research. Section 1 demonstrates the variety of languages present in a single system which are conflated in a DTL, section 2 discusses that such conflation should by necessity apply to notions of packaging, and section 3 concludes with proposed work to ensure that happens.

1.1 A Language Has Many Tongues

A programming language is actually many languages working together.

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

Let us continue using the `C` language for our examples since it is so ubiquitous and has influenced many languages. Such a choice has the benefit of referring to a concrete language, rather than speaking in vague generalities. Besides Agda –a language mentioned throughout the proposal– we shall also refer to Haskell as a representative of the functional side of programming. For example, in Haskell there is no distinction between values and statements —the latter being a particular instance of the former— and so it uses the same notation `if_then_else_` for both. However, in practice, statements in Haskell are more pragmatically used as a body of a `do` block for which the rules of conditionals and local variables change –hence, Haskell is not as uniform as it initially appears.

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

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

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

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

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

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

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

1.2 Needless Distinctions for Containers

Computing is compositionality. Large mind-bending software developments are formed by composing smaller, much more manageable, pieces together. How? In the previous section we outlined a number of languages equipped with term constructors, yet we did not indicate which were more primitive and which could be derived.

The methods currently utilised are ‘ad hoc’, e.g., “dump the contents of packages into a new \"uber package”. What about when the packages contain conflicting names? “Make an uber package with field names for each package’s contents”. What about viewing the new uber package as a hierarchy of its packages? “Make conversion methods between the two representations.” —This *should be* mechanically derivable.

In general, there are special-purpose constructs specifically for working with packages of “usual”, or “day-to-day” expression- or statement-level code. That is, a language for working with containers whose contents live in another language. This forces the users to think of these constructs as rare notions that are rarely needed —since they belong to an ephemeral language. They are only useful when connecting packages together and otherwise need not be learned.

When working with mutually dependent modules, a simple workaround to cyclic type-checking and loading is to create an interface file containing the declarations that dependents require. To mitigate such error-prone duplication of declarations, one may utilise literate programming to tangle the declarations to multiple files —the actual parent module and the interface module. This was the situation with Haskell before its recent module signature mechanism [Kil+14]. Being a purely functional language, it is unsurprising that Haskell treats nested record field updates awkwardly: Where a C-like language may have `a.b.c := d`, Haskell requires `a { b = b a {c = d}}` which necessarily has field names `b, c` polluting the global function namespace as field projections. Since a record is a possibly deeply nested list of declarations, it is trivial to flatten such a list to mechanically generate the names `‘‘a-b-c’’` —since the dot is reserved— unfortunately this is not possible in the core language thereby forcing users to employ ‘lenses’ to generate such accessors by compile-time meta-programming. In the setting of DTLs, records in the form of nested Σ-types tend to have tremendously poor performance —in existing implementations of Coq [GCS14] and Agda [Per17], the culprit generally being projections. More generally, what if we wanted to do something with packages that the host language does not support? “Use a pre-processor, approximate packaging at a different language level, or simply settle with what you have.”

Main Observation Packages, modules, theories, contexts, traits, typeclasses, interfaces, what have you all boil down to dependent records at the end of the day and *really differ* in *how* they are used or implemented. At the end of section 3 we demonstrate various distinct presentations of such notions of packaging arising from a single package declaration.

1.3 Proposed Contributions

The proposed thesis investigates the current state of the art of grouping mechanisms —sometimes referred to as modules or packages—, their shortcomings, and a route to implementing candidate solutions based upon a dependently-typed language.

The introduction of first-class structuring mechanisms drastically changes the situation by allowing the composition and manipulation of structuring mechanisms within the language itself. Granted, languages providing combinators for structuring mechanisms are not new; e.g., such notions already exist for Full Maude [DM07] and B [BGL06]. The former is closer in spirit to our work, but it differs from ours in that it is based on a *reflective logic*: A logic where certain aspects of its metatheory can be faithfully represented within the logic itself. It may well be that the meta-theory of our effort may involve reflection, yet our distinction is that our aim is to form powerful module system features for Dependently-Typed Languages (DTLs).

To the uninitiated, the shift to DTLs may not appear useful, or at least would not differ much from existing approaches. We believe otherwise; indeed, in programming and, more generally, in mathematics, there are three —below: 1, 2a, 2b— essentially equivalent perspectives to understanding a concept. Even though they are equivalent, each perspective has prompted numerous programming languages; as such, the equivalence does not make the selection of a perspective irrelevant. The perspectives are as follows:

1. "Point-wise" or "Constituent-Based": A concept is understood by studying the concepts it is "made out of". Common examples include:

   ◇ A mathematical set is determined by the elements it contains.

   ◇ A method is determined by the sequence of statements or expressions it is composed from.

   ◇ A package —such as a record or data declaration— is determined by its components, which may be *thought of* as fields or constructors.

   Object-oriented programming is based on the notion of inheritance which informs us of "has a" and "is a" relationships.

2. "Point-free" or Relationship Based: A concept is understood by its relationship to other concepts in the domain of discourse. This approach comes into two sub-classifications:

   (a) "First Class Citizen" or "Concept as Data": The concept is treated as a static entity and is identified by applying operations *onto it* in order to observe its nature. Common examples include:

      ◇ A singleton set is a set whose cardinality is 1.

      ◇ A method, in any coding language, is a value with the ability to act on other values of a particular type.

      ◇ A renaming scheme to provide different names for a given package; more generally, applicative modules.

   (b) "Second Class Citizen" or "Concept as Method": The concept is treated as a dynamic entity that is fed input stimuli and is understood by its emitted observational output. Common examples include:

      ◇ A singleton set is a set for which there is a unique mapping to it from any other set. Input any set, obtain a map from it to the singleton set.

- ◇ A method, in any coding language, is unique up to observational equality: Feed it arguments, check its behaviour. Realistically, one may want to also consider efficiency matters.
- ◇ Generative modules as in the `new` keyword from Object oriented programming: Basic construction arguments are provided and a container object is produced.

Observing such a sub-classification as distinct led to traditional structural programming languages, whereas blurring the distinction somewhat led to functional programming.

A simple selection of equivalent perspectives leads to wholly distinct paradigms of thought. It is with this idea that we propose an implementation of first-class grouping mechanisms in a dependently typed language —theories have been proposed, on paper, but as just discussed actual design decisions may have challenging impacts on the overall system. Most importantly, this is a *requirements driven* approach to coherent modularisation constructs in dependently typed languages.

Later on, we shall demonstrate that with a sufficiently expressive type system, a number of traditional programming notions regarding 'packaging up data' become conflated —in particular: Records and modules; which for the most part can all be thought of as "dependent products with named components". Languages without such expressive type systems necessitate certain constraints on these concepts according to their intended usage —e.g., no multiple inheritance for Java's classes and only one instance for Haskell's typeclasses. It is not clear whether such constraints have been brought to more expressive languages out of necessity, convention, or convenience. Hence we propose a systematic exploration of the structuring-mechanism design space for DTLs as a starting point for the design of an appropriate dependently-typed module system. Along the way, we intend to provide a set of atomic combinators that suffice as building blocks for generally desirable features of grouping mechanisms, and moreover we intend to provide an analyses of their interactions.

That is, we want to look at the edge cases of the design space for structuring-mechanism *systems*, not only what is considered 'convenient' or 'conventional'. Along the way, we will undoubtedly encounter 'useless' or non-feasible approaches. The systems we intend to consider would account for, say, module structures with intrinsic types —hence treating them as first class concepts— so that our examination is based on sound principles.

Understandably, some of the traditional constraints have to do with implementations. For example, a Haskell typeclass is generally implemented as a dictionary that can, for the most part, be inlined whereas a record is, in some languages, a contiguous memory block: They can be identified in a DTL, but their uses force different implementation methodologies and consequently they are segregated under different names.

In summary, the proposed research is to build upon the existing state of module systems [DCH03] in a dependently-typed setting [Mac86] which is substantiated by developing an extension to a compiler. The intended outcomes include:

1. A clean module system for DTLs that treats modules uniformly as any other value

type.

2. A variety of use-cases contrasting the resulting system with previous approaches.

3. A module system that enables rather than inhibits efficiency.

4. Demonstrate that module features traditionally handled using meta-programming can be brought to the data-value level; thereby not actually requiring the immense power and complexity of meta-programming.

Most importantly, we intend to implement our theory to obtain validation that it 'works'.

1.4 Overview of the Remaining Chapters

When a programming languages does not provide sufficiently expressive primitives for a concept —such as typeclass derivation [BLS18]— users use some form of pre-processing to accomplish their tasks. In our case, the insufficient primitives are regarding the creation and manipulation of theories —i.e., records, classes, packages, modules. In section 3, we will demonstrate an undisciplined prototype that clarified the requirements of our envisioned system. Even though the prototype appears to be metaprogramming, the aim is not to force users interested in manipulating packages to worry about the intricacies of representations; that is, the end goal is to avoid metaprogramming —which is an over-glorified form of preprocessing. The goal is to *use a dependently-typed language to implement the 'missing' module system features directly inside the language.*

The remainder of the thesis proposal is organised as follows.

- ◇ Chapter II discusses what is expected of modularisation mechanisms, how they could be simulated, their interdefinability in Agda, and discuss a theoretical basis for modularisation.
- ◇ Chapter III outlines missing features from current modularisation systems, their use cases, and provides a checklist for a candidate module system for DTLs.
- ◇ Chapter IV discusses issues regarding implementation matter and the next steps in this research, along with a proposed timeline.
- ◇ Chapter V outlines the intended outcomes of this research effort.

An important design decision is whether the resulting development is intended to be reasoned about or not. If reasoning is important, then a language that better supports it is ideal. That is why we are using Agda –using a simpler language and maintaining data invariants eventually becomes much harder [LM13].

Let us conclude by attempting to justify the title of this thesis proposal.

Landin's *The Next 700 Programming Languages* [Lan66] inspired a number of works, including [BPT17; Pau93; FMP15; Lei07; FMW10] and more. The intended aim of the thesis is a requirements driven approach to coherent modularisation constructs in DTLs. In particular, we wish to extend Agda to be powerful enough to implement the module system features, in the core language, that people actually want and currently mimic by-hand or using third-party preprocessors. An eager fix would be to provide metaprogramming features, but unless one is altering the syntax or producing efficient code, this is glorified pre-processing —it is a means to fake missing abstraction features. Moreover, metaprogramming would be a hammer too big for the nail we are interested in; so big that its introduction might ruin the soundness of the DTLs —e.g., two terms may be ill-typed and ill-formed, such as `x +` and `5 = 3`, but are meaningful when joined together, as in `x + 5 = 3`. Our aim is to provide just the right level of abstraction so that, if anything, users can write a type of container or method upon it then derive '700' simple alternate views of the same container and method.

To be clear, consider a semi-ring —or any simple record of 17 different kinds of data. A semi-ring consists of two monoids —each consisting of a total of 7 items of data and proof matter— where one of them is commutative and there are two distributivity axioms. Hence, a semi-ring consists of 17 items. If we wanted to expose, say, 3 such items —for example, the shared carrier and the identities of each monoid— then there are a total of $\binom{17}{3} = 680$ ways, and if we jump to 4 items we have $\binom{17}{4} = 2380$ possible forms. Of course these numbers are only upper bounds when record fields depend on earlier items. In section 3, we provide explicit examples of different structural presentations of packages.

Usually, library designers provide one or two views, along with conversion functions, and commit to those; instead we want to liberate them to choose whatever presentation is convenient for the tasks at hand and to work comfortably with the guarantee that all the presentations are isomorphic. Humans should be left to tackle difficult and interesting problems; machines should derive the tedious and uninteresting —even if it's simple, it saves time, is less error-prone, and clearly communicates the underlying principle.

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

Chapter 2

Current Approaches

Structuring mechanisms for proof assistants are seen as tools providing administrative support for large mechanisation developments [RS09a], with support for them usually being conservative: Support for structuring-mechanisms elaborates, or rewrites, into the language of the ambient system's logic. Conservative extensions are reasonable to avoid bootstrapping new foundations altogether but they come at the cost of limiting expressiveness to the existing foundations; thereby possibly producing awkward or unusual uses of linguistic phrases of the ambient language.

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

In section 2.1 we define modularisation; in section 2.2 we discuss how to simulate it, and in section 2.3 we review what current systems can and cannot do; then in section 2.4 we provide legitimate examples of the interdefinability of different grouping mechanisms within Agda. We conclude in section 2.5 by taking a look at an implementation-agnostic representation of grouping mechanisms that is sufficiently abstract to ignore any differences between a record and an interface but is otherwise sufficiently useful to encapsulate what is expected of module systems. Moreover, besides looking at the current solutions, we also briefly discuss their flaws.

2.1 Expectations of Module Systems

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

Namespacing Modules provide new unique local scopes for identifiers thereby permitting de-coupling.

The ability to have multiple files contribute to the same namespace is also desirable for de-coupled developments. This necessitates an independence of module names from the names of physical files —such de-conflation permits recursive modules.

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

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

In particular, packages themselves have types which happen to be packages. This is the case with universal algebra, and OCaml, where 'structures' are typed by 'signatures' —note that OCaml's approach is within the same language, whereas, for example, Haskell's recent retrofitting [Kil+14], of its weak module system to allow such interfacing, is not entirely in the core language since, for example, instantiating happens by the package manager rather than by a core language declaration.

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

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

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

With the metaprogramming inspired approach we are proposing, it is only reasonable that, for example, one be able to mechanically transform a package with a local type declaration into a package with the local declaration removed and a new component added to abstract it. That is, a particular implementation is no longer static, but dynamic.

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

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

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

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

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

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

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

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

2.2 Ad hoc Grouping Mechanisms

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

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

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

Objects: An object can be simulated by having a record structure contain the properties of the class which are then instantiated by record instances. Public class methods are then normal methods whose first argument is a reference to the structure that contains the properties.

Muliple Forms of the Template-Instantiation Duality

Template	has a	**Instance**
$\approx$ class		$\approx$ object
$\approx$ type		$\approx$ value
$\approx$ theorem statement		$\approx$ witnessing proof
$\approx$ specification		$\approx$ implementation
$\approx$ interface		$\approx$ implementation
$\approx$ signature		$\approx$ algebra
$\approx$ logic		$\approx$ theory

Modules: Languages that do not support a module may mimic it by placing “module contents” within a record. Keeping all contents within one massive record also solves the namespacing issue.

In JavaScript, for example, a module is a JSON literal —i.e., a comma separated list of key-value pairs. Moreover, encapsulation is simulated by having the module be encoded as a function that yields a record which acts as the public contents of the module, while the non-returned matter is considered private. Due to JavaScript’s dynamic nature we can easily adjoin functionality to such ‘modules’ at any later point; however, we cannot access any private members of the module. This inflexibility of private data is a heavy burden in an Object Oriented Paradigm.

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

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

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

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

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

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

2.3 Existing Systems

We want to implement solutions in a dependently typed language. Let us discuss which are active and their capabilities.

Dependent-types provide an immense level of expressivity thereby allowing varying degrees of precision to be embedded, or omitted, from the type of a declaration. This overwhelming degree of freedom comes at the cost of common albeit non-orthogonal styles of coding and compilation, which remain as open problems that are only mitigated by awkward workarounds such as Coq's distinction of types and propositions for compilation efficiency. The difficulties presented by DTLs are outweighed by the opportunities they provide [AMM05] —of central importance is that they blur distinctions between usual programming constructs, which is in alignment with our thesis.

To the best of our knowledge, as confirmed by Wikipedia in [18d; 18b], there are currently less than 15 *actively developed* dependently-typed languages in-use *that are also used* as proof-assistants —which are intersting to us since we aim to mechanise all of our results: Algorithms as well as theorems.

Agda [BDN; Nor07]: One of the more popular proof assistants around; possibly due to its syntactic inheritance from Haskell —as is the case with Idris. Its Unicode mixfix lexemes permit somewhat faithful renditions of informal mathematics; e.g., calculational proofs can be encoded to be read by those unfamiliar with the system. It also allows traditional functional programming with the ability to 'escape under the hood' and write Haskell code. The language has not been designed solely with theorem proving in mind, as is the case for Coq, but rather has been designed with dependently-typed programming in mind [Jef13; WK18].

The current implementation of the Agda language 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. This change in perspective allows Agda records to act as Haskell typeclasses.

However, the relationship with Haskell is only superficial: Agda's current implementation does not support sharing. In particular, a parameterised module is only syntactic sugar such that each member of the module actually obtains a new functional parameter; as such, a computationally expensive parameter provided to a module invocation may be intended to be computed only once, but is actually computed at each call site.

Coq [Pau; GCS14]: Unquestionably one of, if not, the most popular proof assistant around. It has been used to produce mechanised proofs of the infamous Four Colour Theorem [Gon], the Feit-Thompson Theorem [Gon+13a], and an optimising compiler for the C language: CompCert [Com18; KLW14].

Unlike Agda, Coq supports tactics [Asp+] -a brute force approach that renders (hundredfold) case analysis as child's play: Just refine your tactics till all the subgoals are achieved. Ultimately the cost of utilising tactics is that a tactical proof can only be understood with the aid of the system, and may otherwise be un-insightful and so failing to meet most of the purposes of proof [Far18] —which may well be a large barrier for mathematicians who value insightful proofs.

The current implementation of Coq provides the base features expected of any module system. A notable difference from Agda is that it allows "copy and paste" contents of modules using the include keyword. Consequently it provides a number of module combinators, such as `<+` which is the infix form of module inclusion [Coq18a]. 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 [Cla+07; DM07] framework contains a similar yet more comprehensive algebra of modules and how they work with Maude theories. An important aspect of the thesis work will be to actually investigate Maude further and attempt to reproduce and generalise some of the use cases in 'the Maude book' [Cla+07] using a core set of packaging primitives for DTLs —we will return to what such primitives may be in a later section, on preliminary research. The Common Algebraic Specification Language [Ast+02; BM04; Mos04] will also be investigated with the aim of extracting, and generalising, useful module combinators and their properties.

Incidentally, Coq modules are essentially Agda records —which is unsurprising since our thesis states packaging containers are all essentially the same. In more detail, both notions coincide with that of a signature —a sequence of pairs of name-type declarations. Where Agda users would speak of a record instance, Coq users would speak of a module implementation. To make matters worse, Coq has a notion of records which are far weaker than Agda's; e.g., by default all record field names are globally exposed and records are non-recursive.

Coq's module system extends that of OCaml; a notable divergence is that Coq permits parameterised module types —i.e., parameterised record types, in Agda parlance. Such module types are also known as 'functors' by Coq and OCaml users; which are "generative": Invocations generate new datatypes. Perhaps an example will make this rather strange concept more apparent.

Example of Generative Functors

```
-- Coq                            -- Corresponding Agda

Module Type Unit. End Unit.       -- record Unit : Set where
Module TT <: Unit. End TT.        -- tt : Unit; tt = record {}

Module F (X : Unit).              -- module F (X : Unit) where
  End F.                          --     data t : Set where C : t

Module A := F TT.                 -- module A = F tt
Module B := F TT.                 -- module B = F tt

Fail Check eq_refl : A.t = B.t. -- ≠   eq : A.t ≡ B.t ; eq = refl
```

As seen, in Coq the inductive types are different yet in Agda they are the same. This is because Agda treats such parameterised records, or functors, as 'applicative': They can only be applied, like functions.

For simplicity, we may think of generative functor applications `F X` as actually `F X t` where `t` is an implicit tag such as textual position or clock time. From an object-oriented programming perspective, `F X` for a generative functor `F` is like the `new` keyword in Java/C#: A new instance is created which is distinct from all other instances even though the same class is utilised. So much for the esotericity of generative functors.

Unlike Agda, which uses records to provide traditional record types, Haskell-like typeclasses, and even a module perspective of both, Coq utilises distinct mechanisms for typeclasses and canonical structures. In contrast, Agda allows named instances since all instances are named and can be provided where an implicit failed to be found. Moreover, Coq's approach demands greater familiarity with the unifer than Agda's approach.

Idris [Bra11]: This is a general purpose, functional, programming language with dependent types; alongside ATS, below, it is perhaps the only language in this list that can truthfully boast to being general purpose and to have dependent types. It supports both equational and tactic based proof styles, like Agda and Coq respectively; unlike these two however, Idris erases unused proof-terms automatically rather than forcing the user to declare this far in advance as is the case with Agda and Coq. The only (negligible) downside, for us, is that the use of tactics creates a sort of distinction between the activities of proving and programming, which is mostly fictitious.

Intended to be a more accessible and practical version of Agda, Idris implements the base module system features and includes interesting new ones. Until recently, in Agda, one would write `module _ (x : ℕ) where ⋯` to parameterise every declaration in the block "⋯" by the name `x`; whereas in Idris, one writes `parameters (x : ℕ) ⋯` to obtain the same behaviour –which Agda has since improved upon it via 'generalisation': A declaration's type gets only the variables it actually uses, not every declared parameter.

Other than such pleasantries, Idris does not add anything of note. However, it does provide new constraints. As noted earlier, the current implementation of Idris attempts to erase implicits aggressively therefore providing speedup over Agda. In particular, Idris modules and records can be parameterised but not indexed —a limitation not in Agda.

Unlike Coq, Idris has been designed to "emphasise general purpose programming rather than theorem proving" [Idr18; Bra16]. However, like Coq, Idris provides a Haskell-looking typeclasses mechanism; but unlike Coq, it allows named instances. In contrast to Agda's record-instances, typeclasses result in backtracking to resolve operator overloading thereby having a slower type checker.

Lean [Mou+15; Mou16]: This is both a theorem prover and programming language; moreover it permits quotient types and so the usually-desired notion of extensional equality. It is primarily tactics-based, also permitting a `calc`-ulational proof format not too dissimilar with the standard equational proof format utilised in Agda.

Lean is based on a version of the Calculus of Inductive Constructions, like Coq. Lean is heavily aimed at metaprogramming for formal verification, thereby bridging the gap between interactive and automated theorem proving. Unfortunately, inspecting the language shows that its rapid development is not backwards-compatible —Lean 2 standard libraries have yet to be ported to Lean 3—, and unlike, for example, Coq and Isabelle which are backed by other complete languages, Lean is backed by Lean, which is unfortunately too young to program various tactics, for example.

ATS, Applied Type System: This language combines programming and proving, but is aimed at unifying programming with formal specification. With the focus being more on programming than on proving. [ATS18; CX05]

ATS is intended as an approach to practical programming with theorem proving. Its module system is largely influenced by that of Modula-3, providing what would today be considered the bare bones of a module system. Advocating a programmer-centric approach to program verification that syntactically intertwines programming and theorem proving, ATS is a more mature relative of Idris —whereas Idris is Haskell-based, ATS is OCaml-based.

F*: This language supports dependent types, refinement types, and a weakest precondition calculus [F T18]. However it is primarily aimed at program verification rather than general proof. Even though this language is roughly 8 years in the making, it is not mature —one encounters great difficult in doing anything past the initial language tutorial.

F*'s module system is rather uninteresting, predominately acting as namespace management. It has very little to offer in comparison to Agda; e.g., within the last two years, it obtained a typeclass mechanism —regardless, typeclasses can be implemented as dependent records.

Beluga: The distinctive feature and sole reason that we mention this language is its direct support for first-class contexts [Pie10]. A term `t(x)` may have free variables and so

whether it is well-formed or what its type could be depend 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 we have mentioned, and will reiterate a few times, contexts are behaviourally indistinguishable from dependent sums.

A displeasure of Beluga is that, while embracing the Curry-Howard Correspondence, it insists on two syntactic categories: Data and computation. This is similar to Coq's distinction of `Prop` and `Type`. Another issue is that to a large degree the terms one uses in their type declarations are closed and so have an empty context therefore one sees expressions of the form `[ |- t ]` since `t` is a closed term needing only the empty context. At a first glance, this is only a minor aesthetic concern; yet after inspection of the language's webpage, tutorials, and publication matter, it is concerning that nearly all code makes use of empty contexts —which are easily spotted visually. The tremendous amount of empty contexts suggests that the language is not actually making substantial use of the concept, or it is yet unclear what pragmatic utility is provided by contexts, and, in either way, they might as well be relegated to a less intrusive notation. Finally, the language lacks any substantial standard libraries thereby rendering it more as a proof of concept rather than a serious system for considerable work.

Notable Mentions: The following are not actively being developed, as far we can tell from their websites or source repositories, but are interesting or have made useful contributions. 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' [Bal03; KWP99]; unfortunately it is not a dependently-typed language –though DTLs can be implemented in it. Mizar, unlike the above, is based on (untyped) Tarski–Grothendieck set theory which in some-sense has a hierarchy of sets. Like Coq, it has a large library of formalised mathematics [Miz18; NK09; Ban+18]. Developed in the early 1980s, Nuprl [PRL14] is constructive with a refinement-style logic; besides being a mature language, it has been used to provide proofs of problems related to Girard's Paradox [Coq86]. PVS, Prototype Verification System [Sha+01], differs from other DTLs in its support for subset types; however, the language seems to be unmaintained as of 2014. Twelf [PT15] is a logic programming language implementing Edinburgh's Logical Framework [UCB08; Rab10; SD02] and has been used to prove safety properties of 'real languages' such as SML. A notable practical module system [RS09b] for Twelf has been implemented using signatures and signature morphisms. Matita [Asp+06; Mat16] is a Coq-like system that is much lighter [Asp+09]; it is been used for the verification of a complexity-preserving C compiler.

Dependent types are mostly visible within the functional community, however this is a matter of taste and culture as they can also be found in imperative settings, [Nan+08], albeit less prominently.

2.4 Facets of Structuring Mechanisms: An Agda Rendition

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.

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

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

Monoids as Agda Records

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

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

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

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

Monoids as Typeclasses

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

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

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

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

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

Monoids as Dependent Sums

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

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

Of the renditions thus far, the Σ rendering makes it clear that a monoid could have any subpart as a record with the rest being dependent upon said record. For example, if we had a semigroup type, we could have declared `Monoid-Σ = Σ S : Semigroup • Σ Id : Semigroup.Carrier S`. There are a large number of such hyper-graphs, we have only presented a stratified view for brevity. In particular, `Monoid-Σ` is the extreme unbundled version, whereas `Monoid-Record` is the other extreme, and there is a large spectrum in between –all of which are somehow isomorphic; e.g., `Monoid-Record ≅ Σ C : Set • HasMonoid C`. Our envisioned system would be able to derive any such view at will [Ast+02] and so programs may be written according to one view, but easily repurposed for other view with little human intervention.

Instances and their use are as follows. One may realise that `pop-0` proofs as a form of polymorphism —we will return to package former polymorphism when discussing preliminary research.

Instance Declarations

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

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

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

No Monoids Mentioned at Use Sites

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

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

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

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

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.*

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

Agda

```
-- Following proves: Monoid-Record  ≅  Σ Set HasMonoid.

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

from-record-to-usual-type : Monoid-Record → Monoid-Σ
from-record-to-usual-type M =
  let open Monoid-Record M
  in Carrier , Id , _⨾_ , lid , rid , assoc

  {- Essentially moved from record{⋯} to product listing -}
```

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.

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.

Agda

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

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

Notice that this is nothing more than the named fields of `Monoid-Record` squished into six lines. 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.

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

Agda

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

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

Agda

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

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

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

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

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

Agda

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

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

Agda

```
ℕ-pop_m  : ∀(x y : ℕ)  →  x + 0 + y  ≡  x + y
ℕ-pop_m  =   pop-Id_m
```

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

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

Demonstrating the interdefinablity of different notions of packaging cements our thesis that it is essentially utility that distinguishes packages more than anything else. In particular, explicit distinctions have lead to a duplication of work where the same structure is formalised using different notions of packaging. In chapter 3 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.

2.5 Theory Presentations: A Structuring Mechanism

What of the most closely related theoretical work?

Our envisioned effort would support a "write one, obtain many" approach to package formation. We now turn to mentioning how package formers are currently treated formally under the name of 'theory presentations'. It is the aim of this section to attest that the introduction's story is not completely on shaky foundations, thereby asserting that the aforementioned goals of the introduction are not unachievable —and the problems that will be posed in chapter 3 are not trivial.

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

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

Below is a grammar, from [MRK18], for a simple yet powerful module system based on theory (presentations) and theory morphisms –which are merely named contexts and named substitutions between contexts, respectively. Both may be formed modularly by using includes to copy over declarations of previously named objects. Unlike theories which may include arbitrary declarations, theory morphisms `(V : P → Q) :=` δ are well-defined if for every `P`-declaration `x : T`, δ contains a declaration `x = t` where `t` may refer to all names declared in `Q`. Observe that a context is, up to syntactical differences, essentially JavaScript object notation literal. Consequently, the notion of a mixin as described for JSON literals is here rendered as a theory morphism.

Syntax for Dependently Typed λ-calculus with Theories

```
-- Contexts
Γ  ::= ·                       -- empty context
     | x : τ [:= τ’], Γ         -- context with declaration, optional definition
     | includes X, Γ           -- theory inclusion

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

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

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

-- Example theory hierarchy of signatures, abbrevating “(Π x : A • B) = (A → B)”.
, MagmaSig := Carrier : Set,  _⨾_ : Carrier → Carrier → Carrier, .
, MonSig   := includes MagmaSig, Id : Carrier, .
, .
```

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

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

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

  Theories as Types [MRK18] presents the first formal approach that systematically internalises theories into record types. Their central idea is to introduce a new operator `Mod` –read “models of”— that turns a theory T into a type $\mathsf{Mod}\,T$ which *behaves* like a record type.

◇ Operations on grouping mechanisms [CO12].

As mentioned earlier, a theory morphism, also known as a ‘view’, is a map between contexts that implements the interface of the source using utilities of the target; whence results about specific structures can be constructed by transport along views [FGJ92]: A view `V : P → Q` gives rise to a term homomorphism from `P`-terms to `Q`-terms that is type-preserving in that whenever θ`, P ⊢ t : T` then θ`, Q ⊢  t :    T`. Thus, views preserve judgements and, via the propositions-as-types representations, also preserve truth.

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

Φ Extended to Terms

$$\begin{aligned}
\Phi(x) &= x && \text{Provided } x \text{ is an } \mathcal{S}\text{-variable symbol}\\
\Phi\left(f(t_1, \ldots, t_n)\right) &= \beta(f)\left(\Phi\, t_1, \ldots, \Phi\, t_n\right) && \text{Provided } f \text{ is a } n\text{-ary } \mathcal{S}\text{-function symbol}\\
\Phi\left(\mathcal{Q}\, x \;\bullet\; P\right) &= \left(\mathcal{Q}\, x \;\mid\; U\;\, x \bullet\; \Phi(P)\right) && \text{Provided } \mathcal{Q} \text{ is a variable-binder } \forall, \exists, \lambda
\end{aligned}$$

The *Standard Interpretation Theorem* [Far93] provides sufficient conditions for a translation to be an 'interpretation' which transports results between formalisations. It states: A translation is an interpretation provided $\mathcal{S}$-axioms P are lifted to theorems $\Phi(P)$, the universe of discourse is non-empty ($\exists x \bullet U\, x$), and the interpretation of the universe contains the interpretations of the symbols; i.e., for each $\mathcal{S}$-symbol f of arity n,
$\Phi(\forall x_1, \ldots, x_n \bullet \exists y \;\bullet\; f\, x_1 \ldots\, x_{\,n} \,=\, y)$.

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

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

Chapter 3

Solution Requirements

From the outset we have proposed a particular approach to resolving the needless duplication present in current module systems that are utilised in non-dependently-typed languages. Up to this point, we have only discussed how our approach could mitigate certain troubles; such as a difference of perspectives of modules, or of equivalent operations acting on different perspectives of modules. We now turn to discussing, in the following subsections, what it is that is missing from existing module systems, what one actually wants to do with modules, and conclude with a checklist of features that our proposed system should meet in order to be considered usable and adequate as a thesis-level effort.

3.1 Missing Features

Certain mechanically-derivable concepts, such as different perspectives, are needlessly delegated to the user by pedestrian packaging systems. Besides being tedious and error-prone, the inexpressibility of derivates obscures the corresponding general principles underlying them, thus foregoing any machine assistance in ensuring any correctness or safety-ness guarantees. The desire to pursue a more economical yet powerful packaging system follows from our research team's expedited efforts that could have been mechanised . We will only mention two such use cases.

Expressivity:

A common pattern that can be seen, for example, in the Agda standard library, is of a predicate ensuring desirable properties OF its inputs, then of a record containing the inputs as fields along with a proof of said predicate. More concretely, suppose we have a binary predicate named `IsSemi` and the record is named `Semi`; the predicate form allows us to quantify over inputs as in ∀ `x y` → `IsSemi x y` → ⋯, in contrast the latter approach is intrinsic in nature: ∀ `(s : Semi)` → ⋯ —contrast this with a mathematician naturally declaring "let `s` be a semigroup", whereas almost never do mathematicians say "let `x` be a set and y be an operation on it that together constitute a semigroup”.

At a first glance, it does not seem too troublesome to produce the record presentation from the predicate presentation: Simply *repeat all* the inputs under a record declaration along with a proof obligation. However, the word ‘repeat’ already suggests a problem, and ‘all’ suggests another one. What if one desires to utilise the record associated to the predicate by only packaging certain inputs but not others? This is akin to the problem of constructors in object-oriented languages: In Java, for example, one uses overloading to provide a number of user-written constructors for only a few resonable input invocations to construct an object; in contrast, Common Lisp permits optional named arguments, and so in one fell swoop, with one user-written, constructor, provides all possible combinations of constructor invocations —we are aiming at this level of power and flexibility.

Lest it’s unclear, let’s elaborate slightly on the idea. A semigroup is an algebraic structure that models (untyped) compositionality: It consists of a collection of objects of interests called the `Carrier` set, and an operation `_⨾_` to compose existing items to produce new items, and the operation is associative. Below is a spectrum of ways to bundle up such a structure –starting from being completely bundled up all the way to being completely exposed.

A value of “Semigroup0” is an arbitrary semigroup.

```
-- One extreme: Completely bundled up
record Semigroup0 : Set₁ where
  field
    Carrier : Set
    _⨾_     : Carrier → Carrier → Carrier
    assoc   : ∀ x y z → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

A value of “Semigroup1 C” is a semigroup “structure on” type “C.”

```
-- ‘Typeclass’ on a given Carrier
-- Alternatively: Carrier is known as runtime.
record Semigroup1 (Carrier : Set): Set₁ where
  field
    _⨾_   : Carrier → Carrier → Carrier
    assoc : ∀ x y z → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

A value of “Semigroup2 C op” is a “proof” that ‘C’ with ‘op’ forms a semigroup.

```
-- Two items known at run time --c.f., “IsSemi” above.
record Semigroup2
 (Carrier : Set)
 (_⨾_     : Carrier → Carrier → Carrier) : Set where
  field
    assoc : ∀ x y z → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

The other extreme: Completely unbundled.

```
-- A value of “Semigroup3 C op pf” is trivially the empty record, if any,
-- provided ‘pf’ is a proof that ‘C’ forms a semigroup with ‘op’.
-- This type is usualy written “Σ C : Set • Σ _⨾_ : C → C → C • Σ assoc : ⋯”.
record Semigroup3
 (Carrier : Set)
 (_⨾_ : Carrier → Carrier → Carrier)
 (assoc : ∀ x y z → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)) : Set where
  -- no fields
```

Depending on the user’s needs, it may be useful to have one form or another. Unfortunately they are enslaved to the choices of the library designer, or if they deviate then they must produce tedious conversion methods and use them to pad all the library methods for the structures. Even worse, such back and forth conversions will not only be representation shuffling but also wasteful of resources.

For example, every bijective function $f : A \to B$ furnishes its target B with a semigroup structure provided its source A has the structure to begin with. Since the statement mentions the carriers of semigroups, it is only natural to formulate it an prove it using presentation `Semigroup1`.

Elementary Properties of Functions

```
Surjection : ∀{A B : Set} → (A → B) → Set
Surjection {A} {B} f = ∀ (b : B) → Σ a : A • b ≡ f a
-- (Σ a : A • P a) ≈ { (a, proof)  |  a ∈ A ∧ pf is a proof of P(a) }

Injection : ∀{A B : Set} → (A → B) → Set
Injection {A} {B} f = ∀ {x y} →  f x ≡ f y → x ≡ y
```

An Involved Proof That We Would Like to Reuse

```
translate1 : ∀{A B} → (f : A → B) → Surjection f → Injection f
           → Semigroup1 A → Semigroup1 B
translate1 f surj inj AS =
  let
    open Semigroup1 AS

    -- x ⨾' y is obtained by applying f to the ⨾-composition of the pre-images of x and y.
    infix 5 _⨾'_
    _⨾'_ = λ x y → let a0 = proj₁ (surj x); a1 = proj₁ (surj y) in f (a0 ⨾ a1)

    -- f distributes over ⨾ turning it into ⨾'.
    factor : ∀ {a a'} → f a ⨾' f a' ≡ f (a ⨾ a')
    factor {a} {a'} =
               let a  , m  = surj (f a)
                   a' , w  = surj (f a')
               in
               begin
                 f a ⨾' f a'
               ≡⟨ refl ⟩
                 f (a ⨾ a')
               ≡⟨ cong f (cong₂ _⨾_ (inj (sym m)) (inj (sym w)))  ⟩
                 f (a ⨾ a')


    distribute : ∀ {a a'} → f (a ⨾ a') ≡ f a ⨾' f a'
    distribute {a} {a'} = sym (factor {a} {a'})

  in -- Bundle up ⨾' along with a proof of associtivity
    record { _⨾_ = _⨾'_; assoc = λ x y z →
     let
        -- Obtain f-pre-images
        a₀ , x≈fa₀  =  surj x
        a₁ , y≈fa₁  =  surj y
        a₂ , z≈fa₂  =  surj z
     in
      {- Tersely, we rewrite along the pre-images,
         factor f, perform the associativity of ⨾,
         then distribute f and rewrite along the pre-images.
      -}
       begin
         (x ⨾' y) ⨾' z
       ≡⟨ cong₂ _⨾'_ (cong₂ _⨾'_ x≈fa₀ y≈fa₁) z≈fa₂ ⟩
         (f a₀ ⨾' f a₁) ⨾' f a₂
       ≡⟨ cong (_⨾' f a₂) factor ⟩
         f (a₀ ⨾ a₁) ⨾' f a₂
       ≡⟨ factor ⟩
         f ((a₀ ⨾ a₁) ⨾ a₂)
       ≡⟨ cong f (assoc _ _ _)  ⟩
         f (a₀ ⨾ (a₁ ⨾ a₂))
       ≡⟨ distribute ⟩
         f a₀ ⨾' f (a₁ ⨾ a₂)
       ≡⟨ cong (f a₀ ⨾'_) distribute ⟩
         f a₀ ⨾' (f a₁ ⨾' f a₂)
       ≡⟨ sym (cong₂ _⨾'_ x≈fa₀ (cong₂ _⨾'_ y≈fa₁ z≈fa₂))  ⟩
         x ⨾' (y ⨾' z)

  }
```

translate1 is a lengthy proof, we could repeat it, or invoke it. Since duplication with alteration is error-prone and non-generic, we shall aim for the latter.

Conversions are a Nuisance

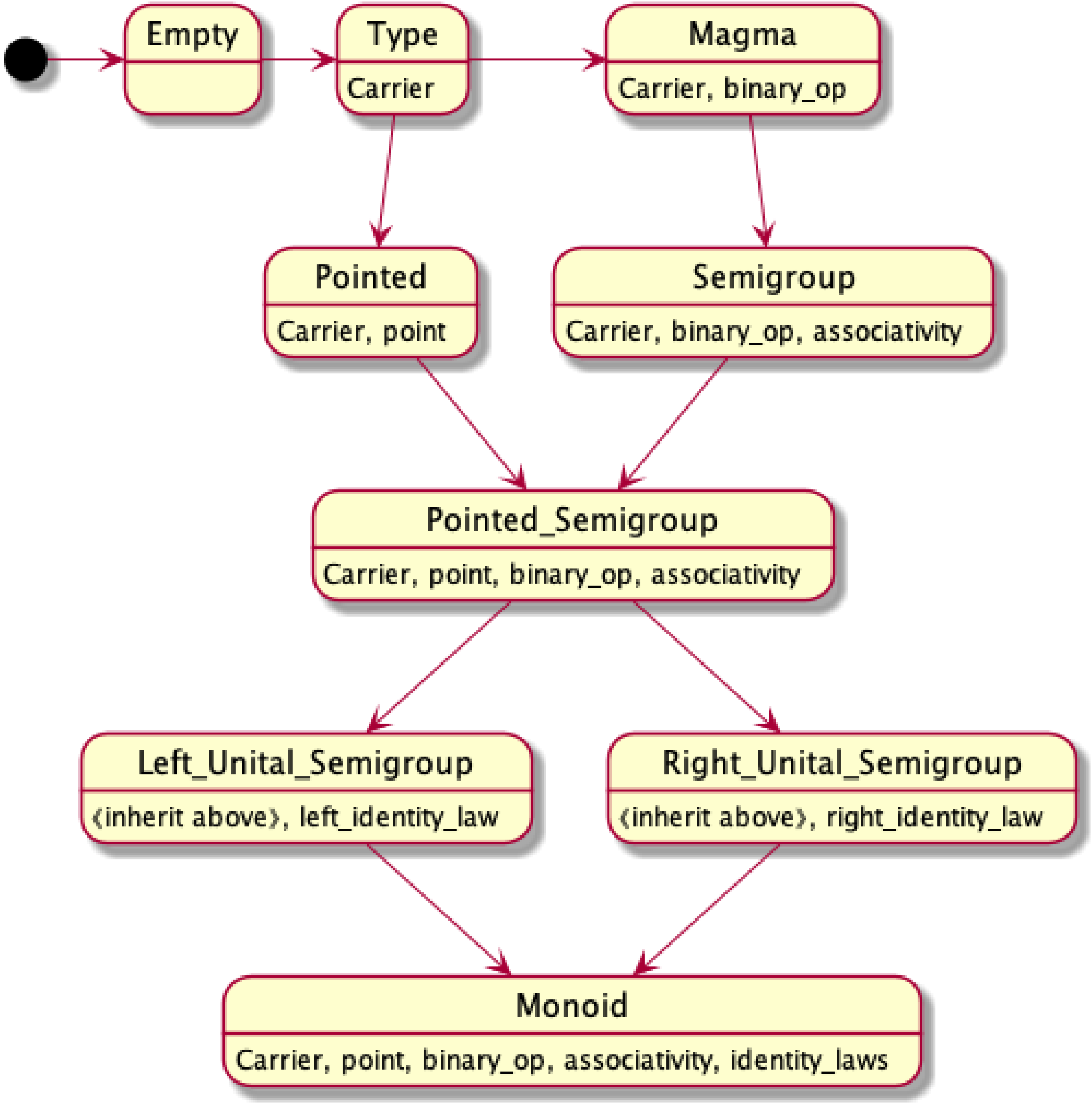
```
translate0 : ∀{B : Set} (AS : Semigroup0) (f : Semigroup0.Carrier AS → B)
           → Surjection f → Injection f
           → Semigroup0
translate0 {B} AS f surj inj = record { Carrier = B ; _⨾_ = _⨾_ ; assoc = assoc }
  where

       -- Repackage ‘AS’ from a ‘Semigroup0’ to a ‘Semigroup1’
       -- only to immediatley unpack it, so that its contents
       -- are available to be repacked above as a ‘Semigroup0’.

       pack : Semigroup1 (Semigroup0.Carrier AS)
       pack = let open Semigroup0 AS
               in record {_⨾_ = _⨾_; assoc = assoc }

       open Semigroup1 (translate1 f surj inj pack)
```

Observe that translate0 repackages AS via pack, then passes that as an argument to translate1, which in turn unpacks it to form a new Semigroup0, which is then unpacked in the last line above. Regardless of any possible wasteful amount of packing and unpacking of records –which may be mitigated via inlining– the way translate0 is written is far from ideal; whereas translate1 is the appropriate level of abstraction to pose the problem. Instead, it would be ideal to write the method at a sufficient level of generality such that translate0 and translate1 are, say, polymorphic instances thereof. This is what we shall propose in a later section.

Excerption:

In order to produce reusable components, theories —i.e., packages— are formed from existing theories by adding only one new concept at a time. Such an approach reduces the possibility of missing a useful structure in the hierarchy, as well as provides tremendous generality —operations can be rendered using the minimal interface required rather than one that is overly expressive. This is a common scheme when formalising mathematics [SW11; GCS14].

Unfortunately, a common scenario is when one wants to *instantiate* such a deeply nested theory. More concretely, suppose we have the following fine-grained hierarchy.


Empty
Type
Carrier
Magma
Carrier, binary_op
Pointed
Carrier, point
Semigroup
Carrier, binary_op, associativity
Pointed_Semigroup
Carrier, point, binary_op, associativity
Left_Unital_Semigroup
⟨inherit above⟩, left_identity_law
Right_Unital_Semigroup
⟨inherit above⟩, right_identity_law
Monoid
Carrier, point, binary_op, associativity, identity_laws

Example Hierarchy

If we have the ingredients for a monoid in hand, we are unfortunately first required to produce a left or right unital semigroup, which requires us to produce a pointed semigroup first, and this regress continues to the base theory, `Type`. Building on semigroups, monoids are a ubiquitous model of compositionality, and so this scenario happens rather often, in one guise or another. The amount of syntactic noise required to produce a simple instantiation is unreasonable: One should not be forced to work through the hierarchy if it provides no immediate benefit. It is to be noted that this issue does not generally apply to implementations of object-oriented class supporting multiple interfaces.

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

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

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

Instead, it would be desirable to 'flatten' the hierarchy into a single package, consisting of the fields throughout the hierarchy, possibly with default implementations, yet still be able to view the resulting package at base levels in the hierarchy. Another benefit of this approach is that it allows users to utilise the package without consideration of how the hierarchy was formed, thereby providing library designers with the freedom to alter it in the future.

These features are considered 'missing' since they are reasonably achievable in a dependently-typed system —e.g., the different forms of dependently-typed bundling suggest a form of polymorphism. Their absence may be due to logistic reasons, such as no effort expedited in their direction, or due to issues surrounding the logical frameworks of the systems. Which is to blame is an investigation matter left to the thesis research.

3.2 Desirable Features

Our preliminary research, and personal use with dependently-typed systems, has yielded three strongly desirable features of a module system for DTLs.

Uniformity:

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

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

Grammar for Terms

```
term ::=  x                     -- variable
      |   f(term_0, ..., term_N) -- function application
```

In pedestrian languages, one distinguishes between *value* terms and *type* terms, whence the t_i are constrained to be homogeneously all values or all types. In contrast, a dependently-typed languages makes no such limitation, thereby allowing the t_i to be heterogeneous. For example, in a simple type system, Maybe (A × List B) is a term where all variables, $t_0, t_1 = A, B$, are of the same kind —types. This is not so with the term
Maybe (A × Vec B n) —A and B are types while n is a number. Our aim is not to educate the reader on the power and utility of dependent types; we invite the reader to consult any of the existing material [AMM05; BDN].

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

Genericity:

Type polymorphism permits us to produce functions written once with type variables and have them applied to radically different types. Likewise, it would be desirable to write once a generic function on a kind of package and have it operate on the many variations of packaging.

An example of this idea is presented at the end of this section, as part of preliminary research. In particular, we demonstrate a novel form of generic programming, *package polymorphism*: A method is written against a generic notion of container and is then applied to derived notions —such as the Semigroup$i$ forms from the previous section.

Extensiblity:

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

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

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

It would be desirable to allow packages to be treated as first-class concepts that could be acted upon, in order to avoid third-party tools that obscure generic operations and leave them out of reach for the powerful typechecker of a dependently typed system.

These features are desirable for working with modules, yet raise a number of immediate concerns. For example, uniformity may lead to ambiguous parsing, genericity may lead to inefficient execution, and extensibility borders on meta-programming thereby leaving the realm of types altogether. Possible limitations on these features may result in the thesis efforts to implement them in a dependently-typed system, such as Agda.

3.3 One-Item Checklist for a Candidate Solution

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

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

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

During the research stage of the thesis, some of the sub-goals may be altered radically, dismissed altogether, or new ones brought forth due to implementation considerations. However, the one main goal will remain unchanged as it is how we have chosen to measure the minimal adequacy for a module system for rich settings that include dependent-types.

3.4 Preliminary Research

The homogeneous treatment of structuring mechanisms is herein presented using a prototype developed using the user-friendly Emacs application framework by means of textual expansion, the details of which are largely uninteresting —suffice it to say, the code is tremendously terse. In this section we demonstrates that packaging concepts differ only in their use, leading to a uniform syntax of which first-class records are an instance and so the resulting system is homoiconic in nature. We introduce fictitious syntax, mostly in red, with its intended Agda elaboration in blue —the users write the red and expect it to behave like the blue; no "code generation" transpires.

The reader is advised to remember that the value of a prototype is in the guidance it provides, not the implementation itself nor any of its design decisions —such as using strings in meta-programming scenarios. In other words, for the reader, portions of this section may serve as an exercise in foresight and patience. (A brief demonstration of the prototype may be viewed at https://www.youtube.com/watch?v=NYOOF9xKBz8 .)

The initiated reader will quickly notice that our package formers are just theory presentations —a list of name-type pairs. The chosen phrasing is due to the target audience, DTL programmers. We are not committed to the name, but unlike the overloaded 'module', 'package former' is a good new name without too many meanings. We have not provided full semantics for package formers, but we have provided concrete well-defined elaborations to communicate the intent: A package former is akin to a type former, it is 'incomplete' and does not define a concrete package until a certain tag is provided. It is part of the thesis effort to investigate which features of our proposed package formers break, or become limited, when considered with other language constructs.

The uniformity in syntax reduces the variety of sub-languages in a dependently-typed language by eliminating needless distinctions for notions of containers. The first subsection below addresses syntactic similarity, whereas the second tackles computing similarity, and we conclude with a brief discussion on foundational concerns.

3.4.1 First Observation: Syntactic Similarity for Containers

Since the prototypical notion of packaging is that of records, which are value terms, all, necessarily succeeding, notions of packaging ought to be treated uniformly as value types. Consequently, variations on packaging should only be signalled by necessary keywords, and otherwise should be syntactically indistinguishable. That is to say, a 'variation' is a tag identifying what particular form of module is desired, such as `datatype` for an algebraic data type with the declared fields as constructors, or as `record` to yield a record structure with constituents being the declared fields.

For example, just as `List` is a type-former, we may declare a 'package former':

Our first package former

```
PackageFormer TermP (v : Variation) : Set where
   Var : Int → TermP v
   Add : TermP v → TermP v → TermP v
```

Note that a package former is just a sequence of names with types and, as will be demonstrated later, optional default types. It requires a particular “interpretation” —possibly user-defined—, to produce some notion of package. This is signalled by the `Variation` type, which for brevity contains `data, record, typeclass`, and a few more that we will meet below.

For example, the `data` variation of packaging gives us a free data type.

Free data type: Terms are integer variables and addition of terms

```
TermData = TermP data
{-
≅  data TermData : Set where
     Var : Int → TermData
     Add : TermData → TermData → TermData
-}
```

In the comment above, we indicate how our fictitious syntax is intended to be elaborated into current Agda syntax. Besides syntax, induction principles are also derived: Our envisioned system would be able to derive simple, tedious, uninteresting concepts; leaving difficult, interesting, ones for humans to solve. For this type, below is the dependently typed eliminator, which in a DTL, corresponds to an induction principle.

Free data types also come with an induction principle

```
{-
   term-data-elim : ∀ {} {R : TermData → Set }
                  → (base : (n : Int) → R (Var n))
                  → (ind  : ∀ {s t} → R s → R t → R (Add s t))
                  → (t : TermData) → R t

   term-data-elim base ind (Var n)   = base n
   term-data-elim base ind (Add s t) = ind rs rt
      where rs = term-data-elim base ind s
            rt = term-data-elim base ind t
-}
```

The type of the package former, for now, could simply be `Set` —c.f., the commented-out elaboration which declares `TermData : Set`. However, if we permit a sufficiently small subtyping system, we may find it desirable to have the type of a package former be itself a package former! Moreover, if package former `t` has type package former `t’`, then the user should be able to use `t` at the levels `t : s` without too much overhead, where `s` is any subtype of t with Set being a minimal such subtype. These thoughts are hurried and it is the purpose of the thesis to investigate what is the appropriate route.

It is often the case that one begins working with a record of useful semantic data, but then, say, for proof automation, may want to use the associated datatype for syntax. The latter should be mechanically derivable, and this is what we aim provide with our package formers. We will not delve into the relationship between free data types and how, for example, their associated catamorphism is necessarily also an interpreter —in the programming languages sense. The reader is invited to consult a reference [Shu16].

We shall not discuss polymorphism along variations, the v components above, as it is orthogonal to our immediate goals. For example, TermP could have a field typed TermP (f v) → TermP (g v) → TermP v, where f and g are operations on variations. Nonetheless, this is a feature that one should be aware of.

The remaining items instantiate package formers for the usual common uses. Including notions of records in item 1; an algorithmic sketch underlying the examples of item 1 is presented in item2; union types and external, second-class, modules in item 3; package former polymorphism in item 4; operating on package formers and inheritance in items 5 and 6; then discuss how package formers handle the diamond problem in item 7. Finally, we close in item 8 by discussing a problem not generally found in pedestrian languages and how it is solved using package formers.

1. The Generality of Package Formers —Products

   To demonstrate the generality of the notion of package formers we shall demonstrate how other common forms could be 'derived' from the single declaration above. It is to be noted that for such a small example, such derived code may be taken for granted, however for much larger theories —for example, a "field" comes with more than 20 fields— the ability to derive different perspectives in a consistent fashion is indispensable; especially when the package is refactored. More realistically, a symmetric rig groupoid uses about 212 coherence laws [**rig_computation**], for which case-splitting, to perform proofs, yields over 200 goals thereby making metaprogramming a tempting approach.

   Records

```
-- An instance of  TermRecord should have a carrier type
-- containing the integers, 'Var', and supports some binary operation, 'Add'.
TermRecord = TermP record
{-
≅   record TermRecord  : Set where
      field
        Carrier : Set
        Var     : Int → Carrier
        Add     : Carrier → Carrier → Carrier
-}
```

In the previous and following invocations, the name `Carrier` is a system internal, for now, and can easily be `renamed` —as will be demonstrated later on. For now, we adhere to a single-sorted stance: Unless indicated otherwise, a `Carrier` will always be included. An example of a two-sorted algebraic structure, graphs, is demonstrated at the end of this subsection.

Built-in names, such as `Carrier`, are generally not ideal. For example, a machine may provide the names `FourLeggedFeline` and `CommutativeIdempotentMonoid` where a human may prefer `Cat` and `JoinSemilattice` instead. As such, the resulting system, would accept 'renaming' functions to generate names. For now, we mostly limit such an approach for brevity.

Haskell-style typeclasses —or Scala-like traits

```
TermOn = TermP typeclass
{-
≅   record TermOn (Carrier : Set) : Set where
      field
        Var     : Int → Carrier
        Add     : Carrier → Carrier → Carrier
-}
```

A pair of functions *on* a declared carrier type

```
TermFunctionsOn = TermP tuples
{-
TermFunctionsOn : Set → Set
TermFunctionsOn C = (Int → C) × (C → C → C)
-}
```

Or the carrier is existential

```
TermFunctions = TermP Σ
-- ≅   TermFunctions   =  Σ C : Set  •  Σ Var : Int → C  •  (C → C → C)
```

Let's show a more intricate yet desirable use.

The interface of non-empty lists

```
PointedSemigroup = TermP record hiding (Var) renaming (Add to _⨾_)
                     field
                       Id     : Carrier
                       ⨾-assoc : ∀ x y z → x ⨾ (y ⨾ z) ≡ (x ⨾ y) ⨾ z
{-
≅   record PointedSemigroup  : Set₁ where
      field
        Carrier : Set
        _⨾_     : Carrier → Carrier → Carrier
        Id      : Carrier
        ⨾-assoc : ∀ x y z → x ⨾ (y ⨾ z) ≡ (x ⨾ y) ⨾ z
-}
```

2. Algorithmically Obtaining Elaborated Types We have discussed how the generic package formers elaborate —each blue comment indicates a standalone isomorphic Agda rendition—, as such it should be unsurprising that the constituents of a package former are dependently typed functions *consuming* each concrete variation in its traditional fashion. Let's clarify this idea further.

   Our example package former

   ```
   PackageFormer TermP (v : Variation) : Set where
      Var : Int → TermP v
      Add : TermP v → TermP v → TermP v
   ```

   The 'type' of the first item, for example, is as follows —where `TermP v` is rewritten using the above introduced names for the sake of clarity.

   The types of a constituents of a package former

   ```
   Var : (v : Variation) → Set

   {- Datatype constructor -}
   Var datatype   =  Int → TermData
   {- Dependent projection -}
   Var record     =  (τ : TermRecord) → Int → TermRecord.Carrier τ
   Var Σ          =  (τ : TermFunctions) → Int → proj₁ τ
   {- Parameter of a constraint -}
   Var typeclass  =  ∀{C} {{_ : TermOn C}} → Int → C
   Var tuples     =  ∀{C} → TermFunctionsOn C → Int → C
   ...
   ```

   An initial glance suggests that this is all ad-hoc; let us demonstrate that this is not the case. Suppose there were a method 𝒯 to obtain the user-provided types of constituents; e.g., the given `Var : Int → TermP v` is indistinguishable from `Var : 𝒯 “Var” (TermP v)`.

   Obtaining User-Provided Types —Under the hood

   ```
   Constituent = String -- Draft idea, not ideal.

   -- “A ⟨n⟩→ B  ≈  A → ··· → A → B” with n-many A's.
   _⟨_⟩→_ : Set → ℕ → Set → Set
   A ⟨ zero   ⟩→ B  =  B
   A ⟨ succ n ⟩→ B  =  A → (A ⟨ n ⟩→ B)

   -- Constituents of package formers give rise to “Set ⟨n⟩→ Set” functions.
   𝒯 : {P : PackageFormer} → Constituent P → Set ⟨ arity P ⟩→ Set
   𝒯 “Var” X  =  Int → X
   𝒯 “Add” X  =  X → X → X
   ```

   It is now trivial to reify the above prescription for `Var` in a uniformly fashion —namely, `Var = `*tp*` “Var”`.

Providing User-Facing Types —Under the hood

```
tp : Constituent → Variation → Set
tp c v@datatype  = 𝒯 c (TermP v)
tp c v@record    = (τ : TermP v) → 𝒯 c ((TermP v).Carrier τ)
tp c v@Σ          = (τ : TermP v) → 𝒯 c (proj₁ τ)
tp c v@typeclass = ∀{C} {{_ : TermP v C}} → 𝒯 c C
tp c v@tuples    = ∀{C} → TermP v C → 𝒯 c C
...
```

For example, invoking this approach we find that `Add`, on `TermRecord`'s, is typed *tp* `‘‘Add’’ record`, which may be rewritten as `(τ : TermRecord) → TermRecord.Carrier τ → TermRecord.Carrier τ → TermRecord.Carri` τ. That is, as expected, `Add` on records consumes a record value then acts as a binary operation on the carrier of said record value. Likewise, we invite the reader to check that `Add` on algebraic datatype `TermData` is typed as a binary constructor.

Users have access to the elaborated types.

Providing User-Facing Types

```
 TermP.Var : ∀{v} → tp ‘‘Var’’ v
 TermP.Add : ∀{v} → tp ‘‘Add’’ v
```

This is particularly useful when one wants to extract such types for re-use elsewhere.

Extracting a single —possibly complicated— signature

```
ListBop = TermP.Add datatype ∘ List
{-
≅  ListBop : Set → Set
   ListBop C = (List C → List C → List C)
-}

ConstrainedBop : (Set → Set) → Set
ConstrainedBop constraint  = TermP.Add typeclass using constraint
{-
≅ ConstrainedBop constraint  =  ∀{C} → constraint C → C → C → C

-- N.B., this would not elaborate without the “using”.
-- Semantically, “P.x y using z = (P.x y)[P v ≔ z]”
-- -the “v” appears from “∀{v}” above.
-}

SetoidBop = TermP.Add record using Setoid
{-
≅ SetoidBop : Setoid ₀ ₀ → Set
  SetoidBop S = Setoid.Carrier C → Setoid.Carrier C → Setoid.Carrier C

-- N.B., this would not elaborate if “Sectoid.Carrier” were undefiend.
-}
```

These examples open a flurry of problems.

At this stage, it is sufficient to have observed what could possibly be performed and that it is not without burden. We will not attempt to clarify any problem nor propose any solution; the thesis effort will contend with these matters further.

3. The Generality of Package Formers —Sums & Modules

   Thus far we have only discussed products; however the proposed general notion of containers should also produce sum types and be used in modules —which are just packages.

   At "least one" of the operations is desired on a declared carrier type

   ```
   TermFunctionsSumOn = TermP sum
   -- ≅  TermFunctionsSumOn C  =  (Int → C)  (C → C → C)
   ```

   In general, this yields a disjoint collection of declarations where each declaration is itself a Σ consisting of the context necessary to ensure that the operations are well-defined.

   For modules,

   Using our package former *within* another package

   ```
     PackageFormer MyDriver (t : TermP record renaming (Carrier to C)) : Set where ···
   -- ≅ module MyDriver (t : TermRecord[Carrier ≔ C]) where ···
   -- ≅ module MyDriver (C : Set) (Var : Int → C) (Add : C → C → C) where ···
   ```

   At least two 'free' invocation notations ought to be supplied:

   (a) `MyDriver t`

   (b) `MyDriver type varOp addOp`

   Multifaceted invocations provide a common use case: No overhead to pack or unpack the constituents of a type former so the sole purpose of an invocation. However, the pragmatic feasibility of such an approach is unclear at this stage.

4. Novel Genericity: 'Package Polymorphism'

   We have a sufficient number of elaborations thus far to demonstrate that the notion of package formers is not without merit. It is now an appropriate moment to address an elephant in the room: *The phrase `TermP v` semantically refers to which type?*

   If `v = datatype` then `TermP v` refers to the associated algebraic datatype. If `v = record`, then there are at least two ways to interpret `TermP v`: As either the record type or as the carrier of a record value. Likewise for other variations. For now, we settle with a monadic-like interpretation: We write `do` τ ← `TermP v;` ··· whenever we wish to refer to the underlying carrier of a concrete package former. Loosely put,

Syntax —Under the hood

```
do τ ← TermP v; b  ≈  v / (λ τ → b)

v@datatype  / f  =  f (TermP v)
v@record    / f  =  ∀(τ : TermP v) → f ((TermP v).Carrier τ)
v@Σ         / f  =  ∀(τ : TermP v) → f (proj₁ τ)
v@typeclass / f  =  ∀{τ} {{_ : TermP v τ}} → f τ
v@tuples    / f  =  ∀{τ} → TermP v τ → f τ
```

The ‘over’ notation, _/_, assumes f is a function acting on types; however, this is not necessary, if the ∀ were replaced with λ, then the result would be a term expression. This is yet another opportunity for investigation during the thesis effort. Moreover, there is the possibility of providing “implicit counterparts” to these variations,; e.g., for tuples one may want ∀{τ} {_ : TermP v τ} → f τ instead, which could be variation, say, tuples-imp. Likewise, we may want notation do-Σ to replace
∀ ··· → ··· with Σ ··· • ···.

Unsurprisingly, this approach subsumes our earlier typing elaboration:
tp c v = do τ ← TermP v; 𝒯 c τ. More concretely, for example, a notion of ‘depth’ for terms may have type ∀ {v} → do τ ← TermP v; (τ → ℕ) —a function that takes a package and yields a number. In the case of v = record, such a function actually takes *two* items: The first being a record value, the second being an element of the carrier of that record value. In the case of v = typeclass, the function takes an argument found by instance search. Likewise, for the remaining variations.

Let us now turn to an example of a function operating on the above many, and all, variations of such packages. This example may appear contrived, yet the power of this form of polymorphism appears at the end of this subsection where one programs towards a *particular* interface and has the result *generalised* to other variations —a prime use case is to code against a typeclass representation and use the same methods on bundled records.

“Times Loop”: Iterate an action n times.

```
-- Suppose I have the following syntactic construction.
repeat : TermData → ℕ → TermData
repeat t Zero       =  Var 0
repeat t (Succ n)   =  Add t (repeat t n)

-- Here is its semantic counterpart.
run : (τ : TermRecord) → TermRecord.Carrier τ → ℕ → TermRecord.Carrier τ
run τ t Zero       =  TermRecord.Var τ 0
run τ t (Succ n)   =  TermRecord.Add τ t (run τ t n)

-- Which is merely multiplication for the naturals.
_×_ : ℕ → ℕ → ℕ
t × Zero      = Zero
t × (Succ n) = t + (t × n)
```

The first two are instances of a package former, and it is not diffcult to construe the naturals as the carrier of a package former. After which, we should be able to write one generic function, by writing according to the pacakge former as the interface.

“Times Loop”: Iterate an action n times.

```
instance
  ℕTerms : TermOn ℕ
  ℕTerms = record {Var = λ n → 0; Add = _+_}

{- IsConsumer is defined below; ignore for now. -}
exp : ∀{v} {{_ : IsConsumer v}}  →  do τ ← TermP v; τ → ℕ → τ
exp t Zero      = Var 0
exp t (Succ n) = Add t (exp t n)
```

For example, we immediately obtain an instance for strings.

“Times Loop”: Iterate an action n times.

```
instance
  STerms : TermOn (List Char)
  STerms = record {Var = λ n → []; Add = _++_}

repeat-s = exp {v = typeclass}
{- Yields a whole family, which includes:

   repeat-s0 : {{TermOn (List Char)}} → List Char → ℕ → List Char
   repeat-s0 c Zero = []
   repeat-s0 c (Succ n) = c ++ repeat c n
-}
```

Now that’s re-use! One function for many semantically distinct types. Notice that invoking `exp` on `ListBop` or `TermFunctionsSumOn` values is ill-typed since the mechanically verifiable constraint `IsConsumer` fails for those variations. Indeed, we may utilise a number of constraints on our package variations, such as the following.

Under the hood constraints

```
data IsConsumer : Variation → Set where
  Prod    : IsConsumer tuples
  DepProd : IsConsumer Σ
  Data    : IsConsumer datatype
  Rec     : IsConsumer record
```

When a user defines a variation, they can signal whether it is a consumer or not. Likewise, one can indicate whether a variation should have `Set`-valued operations on not. Note that a default mechanism could be implemented, but the user should continue to have the ability to enforce a particular discipline —c.f., how `C#` allows the user to enforce the subtyping variance of a type former.

Under the hood constraints

```
data HasConstructiveRelations : Variation → Set where
  Prod    : HasConstructiveRelations tuples
  DepProd : HasConstructiveRelations Σ
  Rec     : HasConstructiveRelations record
```

For example, `data` declarations cannot contain proofs of an arbitrary, but fixed, constructive relation without declaring it as a parameter to the type. Nonetheless, a user may want to be able to express syntactic statements about such proof terms —say for proof automation— and they should have the ability to toggle such a feature.

A more important concern is the type of `exp`: The phrase `do` τ ← `TermP v;` $\tau \to \mathbb{N} \to \tau$ elaborates to different types according to the value of `v`, whence to define `exp` it seems necessary to actually pattern match on it to obtain a concrete type, which, for example, may contain more arguments. Case analysis on the possible packaging variations is far from ideal —one might as well re-implement the definition only on the cases they want rather than all cases. The aim —to be pursued further in the full thesis effort— is to invert the process: *Avoid case analysis in favour of a particularly convenient view.*

This is clarified best by referring to the current prototype language: Lisp. Since all data and methods in a lisp are essentially lists, when one prescribes how to project a value from a possibly nested datatype, then the same prescription essentially directs how to get to the location of that value and so we obtain *generic setters*. The following tiny example demonstrates this idea.

Generic Setters in Lisp

```
(setq xs '("a" nil (x y z) 12))   ;; Heterogenous list of 4 items.
(cadar (cdaddr xs))               ;; ⇒ y
(setf (cadar (cdaddr xs)) 'woah) ;; xs ⇒ '("a" nil (x woah z) 12))
```

It is this flexibility that we aim to provide to users. They code not against a generic variation, but rather along one that is the most appropriate task at hand. We would hope that it would not be unrealistic to then mechanically derive the other forms from it. For example, suppose we wish to define retracts on magmas; rather than define the concept for each possible view, we define it once and obtain it for other views.

Example Algebra

```
PackageFormer MagmaP (v : Variation) : Set where
  _⨾_ : MagmaP v → MagmaP v → MagmaP v

MagmaOn = MagmaP typeclass
AMagma  = MagmaP record
```

The ubiquity of magmas —literally everywhere— lends itself to recall that working with structure, possibly needless structure, may usurp the goals of proof [Far18]: No mathematician would naturally say *let M be an algebra on set C* when it suffices to say

let M be an algebra; yet it may be *convenient* to phrase problems more elegantly when the carrier set is mentioned explicitly [Gar+09]. On the other hand, having the carrier explicit for the sake of typeclass resolution relies on decidable type (non)equality; which may be resonable for a simplly typed language but for a DTL type normalisation generally requires non-trivial, non-constant, computation. Anyhow, as mentioned earlier, bundling data is akin to currying or nesting quantifiers, yet is vastly more expensive since library designers generally commit early to one form or another; in this case
`AMagma ≅ Σ C : Set • MagmaOn C` and
`MagmaOn C ≅ Σ M : AMagma • M.Carrier ≡ C`.

Example Operation

```
retract : ∀{S T} → (f : S → T) → MagmaOn T → MagmaOn S
retract f Tgt = record {_⨾_ = λ x y → f x ⨾ f y} where open MagmaOn Tgt
```

Since `MagmaOn = MagmaP v` where `v = typeclass`, we would ideally be able to derive the generic form —possibly via case analysis.

Variation Generalisation

```
retract-v : ∀{v}
          → ∀ {S T} (f : S → T)
          →  do   tgt ← MagmaP v; tgt ≡ T  -- Intentionally no parens.
          → (do-Σ src ← MagmaP v; src ≡ S)
retract-v = ⋯ -- Unclear at this stage.
```

The record case could, semi-algorithmically, yield:

Verbose Record Case

```
retract-v {record}  :  ∀ {S T} (f : S → T)
                    →  ∀ (Tgt : AMagma) → AMagma.Carrier Tgt ≡ T
                    →  Σ (Src : AMagma) • AMagma.Carrier Src ≡ S
retract-v {record} {S} {T} f Tgt refl =  record { Carrier = S
                                                ;  _⨾_ = λ x y → f x ⨾ f y }
                                       , refl
                                       where open AMagma Tgt
```

From a usability perspective the trivial proofs should not be present and so we need to algorithmically rewrite the above type to omit them, as follows. We would like to preserve the argument syntax, `retract f Tgt`, that was originally declared. Unfortunately, for the record case, the type of `f` must refer to the types of the other magamas if we eliminate the trivial equalities. One possible workaround, as follows, is thus to simply provide a omit the tedious equality proofs since they can be found by instance search.

Usable Record Case

```
retract-v {record}  :  ∀ {S T} (f : S → T)
                    →  ∀ (Tgt : AMagma) {{_ : AMagma.Carrier Tgt ≡ T }}
                    →  proj₁ ({{Σ}} Src : AMagma • AMagma.Carrier Src ≡ S)
retract-v {record} f Tgt  = ⋯

-- “{{Σ}} (x : A) • B x” consists of a pair
-- where the second is found by instance search.
```

Notice that we also project at the end since we do not care about the tedious proof; nor should its existence be forced upon the user.

Before we move on, there is particular reason we have deviated from our `TermP` example to the `MagmaP` concept. The `datatype` variation for `MagmaP` does not provide a way to speak of variables of the data type —indeed `MagmaP datatype` has no closed terms, whence no terms at all. It is thus appropriate to now introduce a variation for syntactic terms *over* some variable set which is then utilised by a mechanically derivable semantic function that is freely homomorphic.

From Syntax to Semantics

```
MagmaTermsOn = MagmaP term-typeclass
{-
≅ data MagmaTermsOn (Vars : Set) : Set where
    Var : Vars → MagmaTermsOn Vars
    _⨾_  : MagmaTermsOn Vars → MagmaTermsOn Vars → MagmaTermsOn Vars

MagmaTermsOn-sem : ∀ {υ} {A}  →  do τ ← MagmaP υ;
                                  (f : A → τ) → MagmaTermsOn A → τ
MagmaTermsOn-sem {record} S f (Var x) = f x
MagmaTermsOn-sem {record} S f (l ⨾ r)  = ll s⨾ rr
  where _⨾s_ = AMagma._⨾_ S
        ll = MagmaTermsOn-sem {record} S f l
        rr = MagmaTermsOn-sem {record} S f r
...
-}
```

We will return to homomorphisms later on, for now it is important to notice that some variations may be useless —as in the empty datatypes. There is also the opportunity to explore co-inductive datatypes.

5. Common Operations on Package Formers It is rather common in the record variation to have multiple instances being mentioned and it is desirable to refer to them with syntactically distinct yet appealing names —such as using subscripts, primes, or other decoration. Moreover, a notion of homomorphism, structure-preservation, can usually be automatically inferred.

   Here we show what such declarations looks like, later we show that such things could be *user defined*.

An example package former

```
PackageFormer TermRelP (v : Variation) : Set where
   Var : Int → TermRelP v
   Add : TermRelP v → TermRelP v → TermRelP v
   Rel : TermRelP v → TermRelP v → Set   -- This time we have a relation as well.
```

A prime-decorated package former

```
Declare PackageFormer TermRelP (v : Variation) decorated (λ x → x ++ "’")
{-
≅ PackageFormer TermRelP’ (v : Variation) : Set where
   Var’ : Int → TermRelP’ v
   Add’ : TermRelP’ v → TermRelP’ v → TermRelP’ v
   Rel’ : TermRelP’ v → TermRelP’ v → Set

-- Coherence Meta-property: ∀ v, d  •  TermRelP v decorated d  ≅  TermRelP v
-}
```

Structure preserving operations

```
Declare Homomorphism TermRelP (v : Variation)
{-
≅ PackageFormer TermRelP-Homomorphism (v : Variation) : Set where

    Src : TermRelP v   decorated  (λ x → x ++ "$_1$")
    Tgt : TermRelP v   decorated  (λ x → x ++ "$_2$")

    map : Src → Tgt
    -- Elaborates to “Carrier Src → Carrier Tgt” in “record” variation.

    var_preservation : ∀ n   → map (Var$_1$ n) ≡ Var$_2$ n
    add_preservation : ∀ x y → map (Add$_1$ x y) ≡ Add$_2$ (map x) (map y)
    rel_preservation : ∀ x y → Rel$_1$ x y → Rel$_2$ (map x) (map y)

NB: The “decorated” annotations are local to the package.
-}
```

6. Inheritance & Defaults for Package Formers

   Things get a bit more interesting with multiple packaging, fields making use of dependent types, and of (multiple) default implementations. Besides defaults, a desirable feature of our envisioned system is the ability to lift definitional extensions into fields of the package, say for more efficient implementations.

Recall our example package former

```
PackageFormer TermP (v : Variation) : Set where
   Var : Int → TermP v
   Add : TermP v → TermP v → TermP v
```

All the pieces of `TermP` but now with additionall new pieces

```
PackageFormer PreOrderedTermP (v : Variation) : Set  inherits-from (TermP v) where
   Ord   : OrderedTermP v → OrderedTermP v → Set
   Refl  : ∀ x → Ord x x
   Trans : ∀ x y z → Ord x y → Ord y z → Ord x z

   -- Two default ‘implementations’

   default₁ Ord x y                =  x ≡ y
   default₁ Refl  x                =  refl
   default₁ Trans _ _ _ refl refl  =  refl

   default₂ Ord x y                =  ⊤
   default₂ Refl  x                =  tt
   default₂ Trans _ _ _ _ _        =  tt
```

Notice how “free type” formation incorporates this new open-ended construct, `Ord`, as a two-value holder. An alternative interpretation would be to eliminate it altogether from the elaborated data declaration. Anyhow, since we elaborate a relation as a pair former, proofs for such a relation cannot be included —otherwise it’s not a “free” type!

Derivied ADT from a package former with constructive relations

```
PreOrderedTermData = PreOrderedTermP data
{-
≅  data PreOrderedTermData : Set where
     Var : Int → OrderedTermData
     Add : PreOrderedTermData → PreOrderedTermData → PreOrderedTermData
     Ord : PreOrderedTermData → PreOrderedTermData → PreOrderedTermData

     -- No reflexitivity axiom on ‘Ord’, nor transitivity!
-}
```

Using a default implementation

```
PreOrderedTermData = PreOrderedTermP data with-default₁
{-
≅  data PreOrderedTermData : Set where
     Var : Int → OrderedTermData
     Add : PreOrderedTermData → PreOrderedTermData → PreOrderedTermData

     -- No ‘Ord’ construction, but instead a constructive relation and properties:

     Ord : PreOrderedTermData → PreOrderedTermData → Set
     Ord x y  =  x ≡ y

     Refl  : ∀ x → Ord x x
     Refl  x  =  refl

     Trans : ∀ x y z → Ord x y → Ord y z → Ord x z
     Trans _ _ _ refl refl  =  refl
-}
```

The naming `Ord, Refl, Trans` could have been altered to refer to the newly declared data type, for simplicity we have avoided such a transformation. Moreover, we could reserve `with-default`$_0$ to simply omit constructive relations from being reified as data constructors.

Keeping the axioms by using a record

```
PreOrderedTermRecord = PreOrderedTermP record
{-
≅   record PreOrderedTermRecord : Set where
      field
        Carrier : Set
        Var     : Int → Carrier
        Add     : Carrier → Carrier → Carrier
        Ord     : Carrier → Carrier → Set
        Refl    : ∀ x → Ord x x
        Trans   : ∀ x y z → Ord x y → Ord y z → Ord x z

     -- Notice that the reflexitivity & transitivity axioms are kept!
-}
```

Moreover, the default implementations means we also have the following declaration, where distinctions are made by the occurenace, or absence, of fields.

Defaults yield additional elaborations

```
{-
    record PreOrderedTermRecord : Set where
      field
        Carrier : Set
        Var     : Int → Carrier
        Add     : Carrier → Carrier → Carrier

      Ord     : Carrier → Carrier → Set
      Ord x y =  x ≡ y

      Refl    : ∀ x → Ord x x
      Refl _ = refl

      Trans   : ∀ x y z → Ord x y → Ord y z → Ord x z
      Trans _ _ _ refl refl = refl
-}
```

Here is our first observation of a uniform presentation of packaging, where the "intended use" differs: Whether we want axioms or not?

Not only is the use amicable, but utilities written for the first elaboration effortlessly apply to instances of the second elaboration. Unfortunately, the relationship is not symmetric —e.g., using the additional information provided by the default implementations, `∀ x y → Ord x y → Add x y ≡ Add y x` is provable for the latter but not the former. As such, there is need to be able to mark results applying to a subtype of a package former, or to eliminate such a desirable feature that reduces needless distinctions when applying utilties of the former to the latter. The thesis will provide a solution with a discussion of the alternatives and why they were not adopted.

7. Package Formers Dispense with The Diamond Problem

   Let's consider combining multiple containers.

A package former for unital magmas

```
Package UnitalTermP (v : Variation) : Set inherits-from (TermP v) where
   unit : UnitalTermP v
   lid  : ∀ x → Add unit x ≡ x
   rid  : ∀ x → Add x unit ≡ x
```

Inheriting from multiple pacakage formers

```
Package PreOrderedMonoid (v : Variation) : Set
      inherits-from (UnitalTermP v; PreOrderedTermP v)
  where
   associative : ∀ x y z → (Add x y) z ≡ Add x (Add y z)
   monotone    : ∀ x x’ y y’ → Ord x x’ → Ord y y’ → Ord (Add x y) (Add x’ y’)
```

This package ought to be indistinguishable from the following, whence allowing tremendously flexible declarations and uses. In particular, there is no longer a need to distinguish between a hierarchical and a flattened perspective, since they are considered identical.

Equivalent backend representation

```
Package PreOrderedMonoid (v : Variation) : Set where

   unitaltermp : UnitalTermP v
   preorderedtermp : PreOrderedTermP v

   associative : ∀ x y z → (Add x y) z ≡ Add x (Add y z)
   monotone    : ∀ x x' y y' → Ord x x' → Ord y y' → Ord (Add x y) (Add x' y')

   -- From which sub-structure does the above “Add” arise?
   --
   -- The “record” and “typeclass” variations elaborate with axioms declaring
   -- that identical names are indeed identical operations:
   carrier_coherence : unitaltermp.Carrier ≡ preorderedtermp.Carrier
   var_coherence     : unitaltermp.Var     ≡ preorderedtermp.Var
   add_coherence     : unitaltermp.Add     ≡ preorderedtermp.Add
   --
   -- They also elaborate with default tedious implementations:
   carrier_coherence = refl; var_coherence = refl; add_coherence = refl

   -- Moreover, we can continue the ‘default’ implementation.
   default₁ monotone _ _ _ _ refl refl = refl
   default₂ monotone _ _ _ _ _ _       = tt
```

8. Package Formers & Representational Shifts

Let us close this section by demonstrating how this genericity can aid in ubiquitous representational shifts that appear rather often in dependently typed programming. In pedestrian languages, there are usually less ways to accomplish a task in dependently typed languages and so programming style is not of great concern. In contrast, in a DTL, a user could, for example, work over an abstract data type where a particular argument is fixed or where it is allowed to vary. The two approaches are a matter of style, but can lead to awkward situations.

More concretely, we consider the bread and buffer of coding: Graphs. Without dependent types we can only speak about graphs *over* a given vertex type, with dependent types we can speak about *a* graph, irrespective of vertex type. The former is tantamount to the context `Vertex : Type` ⊢ `Edges : Vertex → Vertex → Type`, and an empty assumption context ⊢ `Vertex : Set, Edges : Vertex → Vertex → Type` for the latter. However, the latter form sometimes leads us into contexts where we have two graphs `G` and `H` for which we make the tedious constraint `Vertex G ≡ Vertex H`. It would be less clumsy to explicitly declare the two graphs to be *over* the same vertex type.

The previous paragraph mentioned a terse dependently-typed presentation of graphs, let us use the classic presentation as it may be more familiar to readers.

Graph package former

```
PackageFormer GraphP (v : Variation) : Set where
  Vertex, Edges : Set
  src, tgt      : Edges → Vertex

  -- The dependently typed notion of edges.
  derivied
    _⟶_ : Vertex → Vertex → Set
    x ⟶ y  =  Σ e : Edges  •  src e ≡ x  ∧  tgt e ≡ y
```

Graphs as records

```
AGraph = GraphP record renaming (Carrier to ‘‘Vertex’’)
{-
≅    record AGraph : Set where
       field
         Vertex Edges : Set
         src    tgt   : Edges → Vertex
-}

-- NB. The implicitly generated name “Carrier” has been identified with
-- the *declared* name “Vertex”. This is acceptable since they have the same type.
-- Without the identification, the record elaboration would have provided a
-- third type field named “Carrier”.
```

Parameterised graphs as typeclasses

```
GraphOver = TermP typeclass renaming (Carrier to ‘‘Vertex’’)
{-
≅    record GraphOver (Vertex : Set) : Set where
       field
          Edges   : Set
          src tgt : Edges → Vertex
-}
```

With these in hand, our goal is to replace the following first line with the second. However, since both types `GraphOver` and `AGraph` are declared as one liners, such a transition is a cheap as possible.

Parameterised graphs as typeclasses

```
(G H : AGraph) → Vertex G ≡ Vertex H → ⋯

(V : Set) → (G H : GraphOver V) → ⋯
```

In order to *replace a semantic constraint with a syntactic constraint* the user simply need to use a *variant* on packaging. Furthermore, we are ensured
`AGraph ≅ Σ V : Set • GraphOver V.`

Dependently-typed graphs are an curious structure. With a bit of renaming, and adding a few laws, we obtain a ‘setoid’ –i.e., an undirected graph where every node has a self-loop, and paths correspond are essentially edges.

Setoid package former

```
PackageFormer SetoidP (v : Variation) : Set where
  -- Graph structure
  Carrier : Set
  _≈_     : Carrier → Carrier → Set
  -- Properties
  refl  : ∀{e}     → e ≈ e
  sym   : ∀{d e}   → e ≈ d → d ≈ e
  trans : ∀{c d e} → c ≈ d → d ≈ e → c ≈ d
```

A non-dependently-typed ‘signature’ of a structure is generally obtained by discarding the relational operators and all properties. For `SetoidP` one would immediately think the signature consists of just `Carrier`. However, if we view it instead as undirected graphs with self-loops at each node and edge-transitivity, then one would say the signature is the vertices `Carrier` and the edges `_≈_`. It is thus not clear when an item, `_≈_` or `_⟶_`, forms constructive proofs or provides a type family. As such, signature extraction thus requires a parameter identifying which elements constitute ‘proof matter’ —then one simply filters a pacakge-former against this criterion to obtain the associated signature. More generally, this allows us to take an `X` structure and obtain may of its the associated views about where knowledge is consolidated [CFK14], including:

Setoid package former

```
X          = ⟨ Carrier; Operations; Properties ⟩      -- C.f., SetoidP
XOver C    = ⟨ Operations; Properties ⟩
IsX C Ops  = ⟨ Properties ⟩
XSig       = ⟨ Carrier; Operations⟩                   -- C.f., GraphP
```

Having the signature in hand, one can easily and mechanically generate many derivied concepts. For example, a ‘homomorphism’ is a family of functions of the underlying sorts such that the given operations are preserved. Likewise, equality of homomorphisms is extensional equality of the underlying maps. One can then generate closed and open terms and their interpretation functions. With this approach to signature extraction, we can use the same algorithms for the production of, say homomorphisms or other constructs, on completely different algebraic structures, whether they be monoids or graphs. Moreover, this implies that concepts generally not considered for a class of algebras can easily be derived and experimented with; likewise for exploring new algebraic theories. These matters are an application, rather than a goal, of our envisioned system.

The curiosity of graphs is that they are one of the simplest *two-sorted* structures and one of the most common in computing. Counter to intuition, existing packaging systems, namely canonical structures and typeclasses, are oriented toward having a distinct parameter: They cannot work well with multi-parameters; like classical single-sorted algebra. However, the both *aim to solve a usability problem: Having to spell out* *everything is too tedious.* Typeclasses are essentially dictionary look-up, having unicity as an issue. Whereas canonical structures require familiarity with how unifer works –we provide enough information to the unifer to find the desired structure– but, in general, canonical structures do not scale. It is one of the thesis efforts to ensure the the unionised approach scales by a complex example with clear avenues of extension.

It should be clear from these examples that package formers provide expectant generality, including the common uses one is mostly interested in. What about unexpected uses? What if a user wishes to utilise a representation we did not conceive of? They should be able to use the existing language to form it.

3.4.2 Second Observation: Computing Similarity for Containers

By necessity of the first corollary, we are forced to utilise a uniform language between the varying notions of packaging thereby relegating their treatment to be a normal aspect of a language's core vernacular, rather than an extra-linguistic feature. The previous examples hint at possible issues regarding well-definedness of certain constructs. Moreover, we only elaborated on a few compositional operations, `inherits-from, renaming, decorated`, yet users may well wish to utilise their own compositional schemes and so it is imperative that we allow them such a flexibility. Consequently, users ought to be able to define their own compositional mechanisms, thereby necessitating that they be able to manipulate package declarations themselves which in-turn forces the language to be somewhat homoiconic. Moreover, to avoid a hierarchy of languages, the facility for manipulating package declarations must itself be a part of the core language, rather than an extra-linguistic feature —c.f., Coq's Ltac.

In our envisioned setup, every `PackageFormer` declaration adds a clause to a special function,

Under the hood

```
packageInfo : PackageFormer → PackageInfo
packageInfo = compiler defined
```

Where a `PackageInfo` consists of `Name`, which is a list of parameter names and types, along with the name of the package former; and `Declarations`, a list of name-type pairs whose last element is the target type.

PackageInfo: Just another package —for “signatures”

```
{- Draft: Lots of string manipulation, not ideal. -}
record PackageInfo : Set where
  field
    Name         : List (String × String) × String
    Declarations : List (String × List String)
--
-- This is just another package,
-- it incidentally happens to be the representation of packages!
```

It is to be noted that there is no commitment to a string-based representation. It is only a prototype and the thesis will likely move to a better typed representation —otherwise, we may run into too many problems of ill-formed package formers.

Recall our example package former

```
PackageFormer TermP (v : Variation) : Set where
  Var : Int → TermP v
  Add : TermP v → TermP v → TermP v
```

The above declaration provides, under the hood, the following clause to `packageInfo`.

Under the hood

```
packageInfo TermP = record { Name         = ["v", Variation] , "TermP"
                           ; Declarations = [ ("Var", ["Int", "TermP v"])
                                            , ("Add", ["TermP v", "TermP v", "TermP v"])
                                            ]
                           }
```

We are now in a position to provide the semantics for the keyword `Declare`, from the previous section. It takes a `PackageInfo` and declares a `PackageFormer`. There should be a compile-time warning if such declarations are meaningless, ill-formed.

For example, the previous `Declare PackageFormer TermRelP (v : Variation) decorated (λ x → x ++ "’")` can thus be obtained by a user by defining `decorated` as an operation on packages!

User-defined composition scheme

```
_decorated_ : PackageInfo → (String → String) → PackageInfo
pk decorated f = record { Name         = bimap id f pk.Name
                        ; Declarations = fmap (bimap f id) pk.Declarations
                        }
```

To rectify the seemingly wild mixfix notions, we request from the compiler the following suitably general syntactic sugar. An operation, call it, `altered-by` of the type `PackageInfo → List PackageInfo → List X → PackageInfo` automatically obtains the syntactic sugar `p altered-by (q0; ...; qk) with (f0; ...; fN)` —c.f., the `inherits-from` syntax above.

With such terse functional programs for forming composition schemes, there is no need to build much into the compiler.

Users can define other similar operations, such as `decorated-rounded` which replaces the first two binary relations' names with $\subseteq$ and $\subset$; or `decorated-square` to make the renamings $\sqsubseteq$ and $\sqsubset$. Additionally, such renames would propagate into any axioms or derived laws. Moreover, the flexibility to invoke such operations in complex ways allows for intricate renamings to be generated at tremendous scale without worry that future renames would need to be made if the orginal packages included new items. Numerous examples of such renaming transpire manually in the impressive RATH [Kah18] development, as well as in Agda's standard library.

When working with multiple values of the same record type, for example, one encounters a usability problem: Refereeing to the constituents without being verbose. The simplest solution is to qualify each invocation, as in `instance.field`, however this is rather cumbersome, inelegant, and is awkward for mixfix names. An alternative is to locally rename the fields according to a scheme reflecting their use. For example, in a produce construction of 5 items, the field names would be renamed to have a subscript number. In a setting of two instances, a user may instead prefer a primed and an undecorated version of field names. Thus far, by hand we have created these tedious subscript and primed renamings, with our envisioned systems, we need no longer worry about such boilerplate.

In nearly the same fashion, a user could have defined the `inherits-from` compositional scheme. Such a scheme may assume that all identically named items have the same types, and crash otherwise. A user could define a better scheme that takes a renaming function, or another function to handle the crash, or simply omitt conflicting names altogether. The examples suggest that many commonly occurring compositional mechanisms [CO12] can be directly provided by a library, rather than by a particular compiler —this includes the ability to hide fragments, expose the largest well-defined fragment, and to combine packages along a given substructure.

Rather than select what we think is best, we can simply provide the general mechanism to the library designer and allow them the freedom to provide their own schemes.

3.4.3 Next Steps

Our brief examples demonstrate that the less design decisions about packaging made by language designers, the more general, applicable, and, most importantly, increased homogeneity in the resulting datatype language without becoming unityped but rather thanks to being dependently-typed. As mentioned in the previous section on existing approaches, one formalism for packages is that of theories and theory combinators; below we thus draw on some problems from theory combinators rendered toward packaging systems.

We have mentioned that the `record` and `typeclass` perspectives solve the common requirement of structures sharing an identical field. Other than that, we have essentially only outlined a general mechanism for declaring packages and compositional schemes, but have not discussed which are the most common and most useful packaging combinators. It is also desirable to discuss the formal properties of such combinators —if anything, to ensure they are sensible and behave as expected. Moreover, which combinators act as a basis for all packaging combinators? Whence their use ensures the resulting composition is well-formed and they could be targeted for optimisations.

To make our approach accessible, the generic package operations are brought to the user rather than baked into the compiler —too great a distance for most users. The `Declare` syntax reifies `PackageInfo`'s into package declarations, but we have not mentioned under what constraints it can actually provide compiler-time, or typechecking-time, errors of ill-formedness. Moreover, how (in)efficient is this process? Could it be extended to work on variable, runtime provided, declarations for refying packages? Perhaps there is a constraint that suffices for the most common cases? Moreover, having observable `PackageInfo`'s being automatically generated for every package declaration renders representation hiding nearly moot.

The proposed approach boarders on meta-programming. Can type erasure and other compiler-specific optimisations be brought into the homoiconic-like setting being pursued here? We have mentioned a few 'built in' variations for packaging; can such a feature be liberated from the compiler and be bent to the users' will? We would need the ability to explain how a package elaborates.

Tremendous flexibility is demanded from the back-end so as to ignore needless distinctions at the users' level. Whereas the practicality is promising, the feasibility of an implementation for such ambiguous parsing [CZ04] is unclear. It is also unclear what effects identifying syntactically distinct items has on, say, normalisation and propositional equality.

The numerous claims and associaited bookkeeping of details pushes us into using a proof assistant, Agda.

Our examples have been 'variation' polymorphic; we have been even more generic by defining `decorated`. What are the limits of programming genericity provided by our scheme? It would unsurprising if this approach yields the next 700 module systems.

Chapter 4

Approach and Timeline

Packages, modules, classes, (dependent) records, (named) contexts, telescopes, theories, specifications —whatever you wish to call them are essential structuring principles that enable modularity, encapsulation, inheritance, and reuse in formal libraries and programs. Moreover, as we have demonstrated, with the exception of use-cases, there are no significant differences between them in a dependently-typed setting, as [MRK18] present a type theoretic calculus of a variant of record types that corresponds to theories.

4.1 Implementation Matter

We will realise our proposal in an existing compiler and so working with it necessitates our implementations to be more than just 'research quality' but actually ready for a broad audience.

Which compiler and for which language?

Since our attention is focused on dependently typed languages within the realm of Martin-Löf's Type Theory [Mar85], Agda [Jef13] is a natural candidate.

Agda is currently one of the most used tools for proof and program experimentation involving dependent types. With its support for mixfix Unicode lexemes, it has become a strong competitor to Coq [Coq18b; PP89; CH88] for both proof construction [ZDG18; Pop18; AHL17; CTB14; SS13; MKJ09] and general program construction [Jef13; Kan12; Set18a; FSS15] —Agda's lack of *syntactic* distinction between programs and propositions, along with its pattern matching utilities in-place of 'tactic sledgehammers' [Abo06], it has also become an attractive language for introducing dependent types and functional programming [Stu16; WK18; Set18b]. With its syntactic similarity to Haskell, many Agda users treat their Agda code as if it were lazy with the `let` and `where` clauses preserving sharing —which is not the case, since such clauses rewrite to top-level functions [Agd18]. Instead, Agda's evaluation strategy is normal order: Function definitions are invoked before arguments are evaluated, but computations of arguments are *not* shared. This is a prime location for efficiency issues since type-checking in a dependently typed language tends to involve evaluation of terms. Surprisingly this has not stopped users from producing large-scale software developments [Kah18; Kan12; Jef13].

Needless to say, a poor choice of elaboration strategy can lead to a loss of sharing —not that Agda has sharing to begin with—, contain too many undesirable side-effects, hinder efficiency, or forgo compile-time optimisations. For example, Agda, as currently implemented using the Glasgow Haskell Compiler (GHC), is a realisation of Martin-Löf's Type Theory (MLTT) that is heralded as both a programming language and proof assistant. Unfortunately MLTT, as many other dependent logics —such as the Calculus of Constructions with inductive types, which underlies both the Coq and Lean proof assistants— does not account for modules, thereby leaving these as consistency-preserving hacks thrown onto the implementation. As mentioned earlier, Agda simply rewrites modules into top level functions with module parameters realised as parameters to the resulting functions. This is an implementation detail and has little impact on theory construction, however, code reuse becomes unreasonably slow due to the loss of sharing that happens when module arguments need to be re-normalised in each function-counterpart. Consequently, only a minor subset of the Agda community actually *executes* their programs. The rest of the community is generally content with type checking only; which does not hinder the reliability of proof.

It is important to note that we employ Agda only as a proof-of-concept for our proposed exploration of first-class structuring-mechanisms in dependently typed languages. Admittedly Agda's support for Unicode mixfix lexemes makes it a pleasure to work in, with mechanised proofs being little work more than their LaTeX renditions.

4.2 Next Steps

The approach we intend to follow consists of the following steps. Notice that feedback loop of practice into theory.

1. Distill the *true* requirements for a solution; ensure good *fit for purpose* criteria exists.

    ◇ Understand the requirements of 'modularity mechanisms' for DTLs.

    ◇ Narrow down a design by choosing a set of requirements.

    ◇ Identify necessary, and practical, trade-offs. Conflicting feature sets? Usability?

    ◇ Ideally we want our implementations to avoid too much overhead, such as creating an entire new language; this may necessitate the weakening of other functionality.

2. Deepen understanding of the opportunities given by DTL.

- ◇ Understand the relationships between modules, records, contexts, telescopes, and signatures.
  - ○ Do they have differing 'types'?
  - ○ As types themselves, do they have differing 'values'?
  - ○ In the setting of DTLs, are they essentially isomorphic?
  - ○ What are the intended uses? What intentions do particular choices communicate?
    - ⋆ E.g., "$x = y$" communicates an equality and nothing more, whereas "$x \iff y$" communicates a Boolean equality: A redundant, particularised, equality symbol serves to succinctly and elegantly communicate more information.

3. Formulate basic, draft, semantics for a small set of DTL module primitives.

   - ◇ What is the type of a package former?
   - ◇ How does it fit into Agda's existing type hierarchy?
   - ◇ What are the types of the primitives themselves?
     - ○ We wish to avoid metaprogramming after all, and so wish to remain within the language rather than in a metalanguage.

4. Prototype some mechanisms; a combination of old, adapted, and novel ones to demonstrate the power of the system.

   - ◇ Implement the structuring mechanism combinators discussed earlier —such as combination over common-substructures.
     - ○ Possibly begin with reifying first class grouping mechanisms by representing contexts —i..e, sequences of declarations with optional definitions— as records in Agda with the undefined declarations being fields and the rest being derived or definitional.

5. Evaluate the mechanisms —using fit-for-criteria.

   - ◇ Since the realisation would be in Agda, we would keep in touch with the community to ensure that the additions contribute to program design.
   - ◇ Evaluate the strength of the resulting additions in terms of practical use for library designers as well as in terms of program speed.

6. Make sure to have a denotational semantics for the mechanisms.

   - ◇ Ensure that the additions are minimal, orthogonal, and construct a sound type theory around them.

7. Refine 2-6 until elegance, or deadline, is reached, whichever comes first.

Our timeline will discuss how we will carry out this approach in multiple passes and will discuss the conditions of a successful pass.

4.3 Timeline

We shall iterate through the ‘approach phase’ three times, utilising a feedback loop of practice into theory. The phases are discussed below.

As our results will likely need to be re-proven for definitional adjustment, we intend to *mechanise* all of our proofs in Agda as well —when possible. Therefore, Agda plays multiple roles: A dependently-typed language to experiment with, as well as a proof checker for our results.

4.3.1 The First Pass: May-October 2019

This stage concludes successfully provided the following checkpoints are achieved.

- ◇ A thorough understanding of what is being done by others, and how our approach differs, is obtained *and* documented.
- ◇ Understand the Agda compilation ecosystem, provide a report on how to make alterations to it, and actually implement at least one structuring mechanism and provide use cases as well as preliminary efficiency analysis.
- ◇ A publication covering existing mechanisms, their features and flaws, and possibly an explanation of why there is theoretical work on these issues but little to no implementation on them —with a focus on practical uses and possible hurdles to use.
  - ○ A side-effect of this is to produce an evaluation strategy for the mechanisms.
  - ○ Moreover, this necessitates looking into the associated semantics, evaluating them, and proposing semantics for the mechanism we have designed.
- ◇ Thesis writing should have begun and nearing completion are sections on introduction and background.

4.3.2 The Middle Pass: November 2019 - February 2020

This stage concluded successfully provided the following checkpoints are achieved.

- ◇ The success of the previous stage ensures an understanding of the Agda compilation ecosystem, as such it should take less time to implement the more mechanisms, theory combinators. The goal is to have the remaining mechanisms implemented, with a focus on the combination-over-a-structure mechanism.

  - With each implementation, reach-out to the Agda community to solicit feedback regarding improvements and possible use cases.

- Extending the semantics for the newly implemented mechanisms.

  - Evaluating which mechanisms are more primitive, which are derived, and which can be used to *allow users to make their own **using** the concrete language itself*!

- A publication of case studies utilising these combinators, as well as a comparison of how these are an improvement over traditional methods.

  - Analysing the interactions between features; does the addition of one hinder another.
  - Empirical tests for efficiency and utility.

- Thesis writing should have progressed with sections on use cases, semantics, and feature design, having substantial matter if not nearing completion.

4.3.3 The Final Pass: March - April 2020

This stage concluded successfully provided the following checkpoints are achieved.

- Ensure that our implementations are meeting our requirements for a solution.
- Begin mechanisation of proofs authenticating that the denotational semantics has desired, expected, properties; such as soundness and safeness.

4.3.4 Concluding Phase

Wrap up all proof matters and finish the thesis.

Suffice to say life tends to be more hectic than a schedule may permit and as such some times may deviate from the above intentions. Regardless, the goal will be to complete the thesis within 2 years time; in particular before September 2020.

Chapter 5

Conclusion

As already discussed, more often than not a module system is an afterthought secondary citizen whose primary purpose is to act as a namespace delimiter —e.g., C#'s `namespace` construct— while relatively more effort is given to their abstraction encapsulation counterpart, e.g., C#'s `class`'es. Some languages' module systems blend both namespace management and implementation hiding, e.g., as in the Haskell programming language. Other languages such as OCaml take modules even further: Not only are modules used for namespace organisation and datatype abstraction, but they can also be passed around as values for manipulation as if they were nothing special, thereby collapsing the distinction between record constructs and organisational constructs.

The proposed research is to build upon the existing state of module systems and develop an extension to a compiler to substantiate our claims, and to ultimately discover new semantical relationships between programming language constructs in a dependently typed setting with modules as first-class citizens. This involves redesigning and enhancing existing module systems to take into account dependent types as well as producing rewrite theorems to ensure acceptable performance times.

Intended outcomes include:

1. A clean module system for DTLs

   ◇ Dependent types blur many distinctions therefore rendering certain traditional programming constructs as inter-derivable and so only a minimal amount need be supported directly, while the rest can be defined within the extended type theory we will be creating. Since modules are records, which are one-field algebraic data types, and we can form sums of modules, it would not be surprising if first-class modules suffice for arbitrary data type definitions.

2. *Utility Objectives*: A variety of use-cases contrasting the resulting system with previous approaches. In particular, the system should:

- ◇ Reduce amount of 'noise' necessary for working with grouping mechanisms in a number of ways.
- ◇ It should be easy and elegant to use and, possibly, to extend.

3. A module system that enables rather than inhibits (or worse) efficiency.

   - ◇ Currently Agda modules, for example, are sugar for extra functional parameters and so all implicit sharing in modules is lost at compilation time.
   - ◇ Deeply nested, deeply tagged, operations could be costly and so being apply to *soundly* flatten modules and *soundly* extract operations and results is a necessity when speed is concerned —moreover, this needs to be mechanical and succinct if it is to be useful.

4. Demonstrate that module features usually requiring meta-programming can be brought to the data-value level.

   - ◇ Names and types, for example, in a module should be accessible and alterable. For example, we can obtain a rig by combining two instances of a monoid module where we would rename the fields of one, or both, of them.
   - ◇ Thereby relegating abstract syntax tree and programs-as-strings manipulations to the edges of the computing environment.

Most importantly, we intend to implement our theory to obtain validation that it "works"!

Bibliography

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

[18b] *Dependent type — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/wiki/Dependent_type (visited on 10/19/2018).

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

[18d] *Proof assistant — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/wiki/Proof_assistant (visited on 10/19/2018).

[Abo06] Kamal Aboul-Hosn. “A Proof-Theoretic Approach to Tactics”. In: *Mathematical Knowledge Management, 5th International Conference, MKM 2006, Wokingham, UK, August 11-12, 2006, Proceedings*. 2006, pp. 54–66. DOI: 10.1007/11812289_6. URL: https://doi.org/10.1007/11812289%5C_6.

[Agd18] The Agda Team. *Agda’s Documentation*. 2018. URL: https://agda.readthedocs.io/en/v2.5.4.1/index.html (visited on 10/12/2018).

[AHL17] Sergio Antoy, Michael Hanus, and Steven Libby. “Proving Non-Deterministic Computations in Agda”. In: *Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, and 24th International Workshop on Functional and (Constraint) Logic Programming, WLP 2015 / WLP 2016 / WFLP 2016, Dresden and Leipzig, Germany, 22nd September 2015 and 12-14th September 2016.* 2017, pp. 180–195. DOI: 10.4204/EPTCS.234.13. URL: https://doi.org/10.4204/EPTCS.234.13.

[AMM05] Thorsten Alkenkirch, Conor McBride, and James McKinna. *Why Dependent Types Matter*. 2005. URL: http://www.cs.nott.ac.uk/~psztxa/publ/ydtm.pdf (visited on 10/19/2018).

[Asp+] Andrea Asperti et al. *A new type for tactics*. URL: http://matita.cs.unibo.it/PAPERS/plmms09.pdf (visited on 10/19/2018).

[Asp+06] Andrea Asperti et al. “Crafting a Proof Assistant”. In: *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*. 2006, pp. 18–32. DOI: 10.1007/978-3-540-74464-1_2. URL: https://doi.org/10.1007/978-3-540-74464-1%5C_2.

[Asp+09] A. Asperti et al. “A compact kernel for the calculus of inductive constructions”. In: *Sadhana* 34.1 (Feb. 2009), pp. 71–144. ISSN: 0973-7677. DOI: 10.1007/s12046-009-0003-3. URL: http://dx.doi.org/10.1007/s12046-009-0003-3.

[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.

[ATS18] The ATS Team. *The ATS Programming Language: Unleashing the Potentials of Types and Templates!* 2018. URL: http://www.ats-lang.org/#What_is_ATS_good_for (visited on 10/19/2018).

[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.

[Ban+18] Grzegorz Bancerek et al. “The Role of the Mizar Mathematical Library for Interactive Proof Development in Mizar”. In: *J. Autom. Reasoning* 61.1-4 (2018), pp. 9–32. DOI: 10.1007/s10817-017-9440-6. URL: https://doi.org/10.1007/s10817-017-9440-6.

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

[BDN] Ana Bove, Peter Dybjer, and Ulf Norell. “A Brief Overview of Agda — A Functional Language with Dependent Types”. In: pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6.

[BGL06] Sandrine Blazy, Frédéric Gervais, and Régine Laleau. “Reuse of Specification Patterns with the B Method”. In: *CoRR* abs/cs/0610097 (2006). arXiv: cs/0610097. URL: http://arxiv.org/abs/cs/0610097.

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

[BLS18] Baldur Blöndal, Andres Löh, and Ryan Scott. “Deriving via: or, how to turn hand-written instances into an anti-pattern”. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*. 2018, pp. 55–67. DOI: 10.1145/3242744.3242746. URL: https://doi.org/10.1145/3242744.3242746.

[BM04] Michel Bidoit and Peter D. Mosses. *Casl User Manual - Introduction to Using the Common Algebraic Specification Language*. Vol. 2900. Lecture Notes in Computer Science. Springer, 2004. ISBN: 3-540-20766-X. DOI: 10.1007/b11968. URL: https://doi.org/10.1007/b11968.

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

[BPT17] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. “The next 700 syntactical models of type theory”. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. 2017, pp. 182–194. DOI: 10.1145/3018610.3018620. URL: https://doi.org/10.1145/3018610.3018620.

[Bra11] Edwin C. Brady. “IDRIS — Systems Programming Meets Full Dependent Types”. In: *Proceedings of the 5th ACM workshop on Programming languages meets program verification*. PLPV ’11. Austin, Texas, USA: ACM, 2011, pp. 43–54. ISBN: 978-1-4503-0487-0. DOI: http://doi.acm.org/10.1145/1929529.1929536. URL: http://doi.acm.org/10.1145/1929529.1929536.

[Bra16] Edwin Brady. *Type-driven Development With Idris*. Manning, 2016. ISBN: 9781617293023. URL: http://www.worldcat.org/isbn/9781617293023.

[CFK14] Jacques Carette, William M. Farmer, and Michael Kohlhase. *Realms: A Structure for Consolidating Knowledge about Mathematical Theories*. 2014. arXiv: 1405.5956v1 [cs.MS].

[CH88] Thierry Coquand and Gérard P. Huet. “The Calculus of Constructions”. In: *Inf. Comput.* 76.2/3 (1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3. URL: https://doi.org/10.1016/0890-5401(88)90005-3.

[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.

[CO12] Jacques Carette and Russell O’Connor. “Theory Presentation Combinators”. In: *Intelligent Computer Mathematics* (2012), pp. 202–215. ISSN: 1611-3349. DOI: 10.1007/978-3-642-31374-5_14. URL: http://dx.doi.org/10.1007/978-3-642-31374-5_14.

[Com18] The Compcert Team. *The Compcert C Compiler*. 2018. URL: http://compcert.inria.fr/compcert-C.html (visited on 10/19/2018).

[Coq18a] 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.

[Coq18b] The Coq Team. *The Coq Proof Assistant*. 2018. URL: https://coq.inria.fr/ (visited on 10/12/2018).

[Coq86] Thierry Coquand. "An Analysis of Girard's Paradox". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. 1986, pp. 227–236.

[CTB14] Ernesto Copello, Alvaro Tasistro, and Bruno Bianchi. "Case of (Quite) Painless Dependently Typed Programming: Fully Certified Merge Sort in Agda". In: *Programming Languages - 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings*. 2014, pp. 62–76. DOI: 10.1007/978-3-319-11863-5_5. URL: https://doi.org/10.1007/978-3-319-11863-5%5C_5.

[CX05] Chiyan Chen and Hongwei Xi. "Combining programming with theorem proving". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*. 2005, pp. 66–77. DOI: 10.1145/1086365.1086375. URL: http://doi.acm.org/10.1145/1086365.1086375.

[CZ04] Claudio Sacerdoti Coen and Stefano Zacchiroli. "Efficient Ambiguous Parsing of Mathematical Formulae". In: *Mathematical Knowledge Management* (2004), pp. 347–362. ISSN: 1611-3349. DOI: 10.1007/978-3-540-27818-4_25. URL: http://dx.doi.org/10.1007/978-3-540-27818-4_25.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. "A type system for higher-order modules". In: *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*. 2003, pp. 236–249. DOI: 10.1145/640128.604151. URL: https://doi.org/10.1145/640128.604151.

[DJH] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. "A formal specification of the Haskell 98 module system". In: pp. 17–28. URL: http://doi.acm.org/10.1145/581690.581692.

[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.

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

[F T18] The F$^*Team$. F$^*OfficialWebsite$. 2018. URL: https://www.fstar-lang.org/ (visited on 10/19/2018).

[Far18] William M. Farmer. *A New Style of Proof for Mathematics Organized as a Network of Axiomatic Theories*. 2018. arXiv: 1806.00810v2 [cs.LO].

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

[FGJ92] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. "Little theories". In: *Automated Deduction—CADE-11*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 567–581. ISBN: 978-3-540-47252-0.

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

[FMP15] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. "The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations - Part 2 - A Survey". In: *J. Autom. Reasoning* 55.4 (2015), pp. 307–372. DOI: 10.1007/s10817-015-9327-3. URL: https://doi.org/10.1007/s10817-015-9327-3.

[FMW10] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. "The next 700 data description languages". In: *J. ACM* 57.2 (2010), 10:1–10:51. DOI: 10.1145/1667053.1667059. URL: https://doi.org/10.1145/1667053.1667059.

[FSS15] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. "Pi-Ware: Hardware Description and Verification in Agda". In: *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*. 2015, 9:1–9:27. DOI: 10.4230/LIPIcs.TYPES.2015.9. URL: https://doi.org/10.4230/LIPIcs.TYPES.2015.9.

[Gar+09] François Garillot et al. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. Ed. by Tobias Nipkow and Christian Urban. Vol. 5674. Lecture Notes in Computer Science. Munich, Germany: Springer, 2009. URL: https://hal.inria.fr/inria-00368403.

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

[Gon] Georges Gonthier. *Formal Proof–The Four-Color Theorem*. URL: http://www.ams.org/notices/200811/ (visited on 10/19/2018).

[Gon+13a] Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14. URL: https://doi.org/10.1007/978-3-642-39634-2%5C_14.

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

[Hal+] Thomas Hallgren et al. "An Overview of the Programatica Toolset". In: *HCSS '04*. URL: http://www.cse.ogi.edu/PacSoft/projects/programatica/.

[Idr18] The Idris Team. *Idris: Frequently Asked Questions*. 2018. URL: http://docs.idris-lang.org/en/latest/faq/faq.html (visited on 10/19/2018).

[Jef13] Alan Jeffrey. "Dependently Typed Web Client Applications - FRP in Agda in HTML5". In: *Practical Aspects of Declarative Languages - 15th International Symposium, PADL 2013, Rome, Italy, January 21-22, 2013. Proceedings*. 2013, pp. 228–243. DOI: 10.1007/978-3-642-45284-0_16. URL: https://doi.org/10.1007/978-3-642-45284-0%5C_16.

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

[Kan12] Karim Kanso. "Agda as a platform for the development of verified railway interlocking systems". PhD thesis. Swansea University, UK, 2012. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.678306.

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

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. "Formal C Semantics: CompCert and the C Standard". In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. 2014, pp. 543–548. DOI: 10.1007/978-3-319-08970-6_36. URL: https://doi.org/10.1007/978-3-319-08970-6%5C_36.

[KS01] Wolfram Kahl and Jan Scheffczyk. "Named Instances for Haskell Type Classes". In: 2001.

[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.

[Lan66] Peter J. Landin. “The next 700 programming languages”. In: *Commun. ACM* 9.3 (1966), pp. 157–166. DOI: 10.1145/365230.365257. URL: https://doi.org/10.1145/365230.365257.

[Lei07] António Menezes Leitão. “The next 700 programming libraries”. In: *International Lisp Conference, ILC 2007, Cambridge, UK, April 1-4, 2007*. 2007, p. 21. DOI: 10.1145/1622123.1622147. URL: https://doi.org/10.1145/1622123.1622147.

[Ler00] Xavier Leroy. “A modular module system”. In: *J. Funct. Program.* 10.3 (2000), pp. 269–303. URL: http://journals.cambridge.org/action/displayAbstract?aid=54525.

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

[LM13] Sam Lindley and Conor McBride. “Hasochism: the pleasure and pain of dependently typed haskell programming”. In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. 2013, pp. 81–92. DOI: 10.1145/2503778.2503786. URL: https://doi.org/10.1145/2503778.2503786.

[Mac86] David B. MacQueen. “Using Dependent Types to Express Modular Structure”. In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. 1986, pp. 277–286. DOI: 10.1145/512644.512670. URL: https://doi.org/10.1145/512644.512670.

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

[Mat16] The Matita Team. *The Matita Interactive Theorem Prover*. 2016. URL: http://matita.cs.unibo.it (visited on 10/19/2018).

[Miz18] The Mizar Team. *Mizar Home Page*. 2018. URL: http://www.mizar.org/ (visited on 10/19/2018).

[MKJ09] Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. “Algebra of programming in Agda: Dependent types for relational program derivation”. In: *J. Funct. Program.* 19.5 (2009), pp. 545–579. DOI: 10.1017/S0956796809007345. URL: https://doi.org/10.1017/S0956796809007345.

[Mos04] Peter D. Mosses. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*. Vol. 2960. Lecture Notes in Computer Science. Springer, 2004. ISBN: 3-540-21301-5. DOI: 10.1007/b96103. URL: https://doi.org/10.1007/b96103.

[Mou+15] Leonardo Mendonça de Moura et al. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. 2015, pp. 378–388. doi: 10.1007/978-3-319-21401-6_26. url: https://doi.org/10.1007/978-3-319-21401-6%5C_26.

[Mou16] Leonardo de Moura. "Formalizing Mathematics using the Lean Theorem Prover". In: *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2016, Fort Lauderdale, Florida, USA, January 4-6, 2016*. 2016. url: http://isaim2016.cs.virginia.edu/papers/ISAIM2016%5C_Proofs%5C_DeMoura.pdf.

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

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

[Nan+08] Aleksandar Nanevski et al. "Ynot: dependent types for imperative programs". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 2008, pp. 229–240. doi: 10.1145/1411204.1411237. url: http://doi.acm.org/10.1145/1411204.1411237.

[NK09] Adam Naumowicz and Artur Kornilowicz. "A Brief Overview of Mizar". In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. 2009, pp. 67–72. doi: 10.1007/978-3-642-03359-9_5. url: https://doi.org/10.1007/978-3-642-03359-9%5C_5.

[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.

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

[Pau93] Lawrence C. Paulson. "Isabelle: The Next 700 Theorem Provers". In: *CoRR* cs.LO/9301106 (1993). url: http://arxiv.org/abs/cs.LO/9301106.

[Per17] Natalie Perna. *(Re-)Creating sharing in Agda's GHC backend*. Jan. 2017. url: https://macsphere.mcmaster.ca/handle/11375/22177.

[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.

[Pop18] Jeremy Pope. "Formalizing Constructive Quantifier Elimination in Agda". In: *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018.* 2018, pp. 2–17. DOI: 10.4204/EPTCS.275.2. URL: https://doi.org/10.4204/EPTCS.275.2.

[PP89] Frank Pfenning and Christine Paulin-Mohring. "Inductively Defined Types in the Calculus of Constructions". In: *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*. 1989, pp. 209–228. DOI: 10.1007/BFb0040259. URL: https://doi.org/10.1007/BFb0040259.

[PRL14] The PRL Team. *PRL Project: Proof/Program Refinment Logic*. 2014. URL: http://www.nuprl.org (visited on 10/19/2018).

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

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

[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.

[RS09a] Florian Rabe and Carsten Schürmann. "A practical module system for LF". In: *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP '09, McGill University, Montreal, Canada, August 2, 2009*. 2009, pp. 40–48. DOI: 10.1145/1577824.1577831. URL: http://doi.acm.org/10.1145/1577824.1577831.

[RS09b] 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.

[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.

[Set18a] Anton Setzer. "Modelling Bitcoin in Agda". In: *CoRR* abs/1804.06398 (2018). arXiv: 1804.06398. URL: http://arxiv.org/abs/1804.06398.

[Set18b] Anton Setzer. *Teaching Agda*. 2018. URL: http://www.cse.chalmers.se/research/group/logic/AIM/AIM6/SetzerTeachingAgda.pdf (visited on 10/12/2018).

[Sha+01] Natarajan Shankar et al. *PVS Prover Guide*. 2001. URL: http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf (visited on 04/19/2019).

[She] Tim Sheard. "Generic Unification via Two-Level Types and Parameterized Modules". In: *ICFP 2001*. to appear. acm press.

[SHH01] Tim Sheard, William Harrison, and James Hook. "Modeling the Fine Control of Demand in Haskell." (submitted to Haskell workshop 2001). 2001.

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

[SS13] Christopher Schwaab and Jeremy G. Siek. "Modular type-safety proofs in Agda". In: *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*. 2013, pp. 3–12. DOI: 10.1145/2428116.2428120. URL: http://doi.acm.org/10.1145/2428116.2428120.

[Stu16] Aaron Stump. *Verified Functional Programming in Agda*. New York, NY, USA: Association for Computing Machinery and Morgan &#38; Claypool, 2016. ISBN: 978-1-97000-127-3.

[SW11] Bas Spitters and Eelis van der Weegen. "Type classes for mathematics in type theory". In: *Mathematical Structures in Computer Science* 21.4 (2011), pp. 795–825. DOI: 10.1017/S0960129511000119. URL: https://doi.org/10.1017/S0960129511000119.

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

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

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

[ZDG18] Ran Zmigrod, Matthew L. Daggitt, and Timothy G. Griffin. "An Agda Formalization of Üresin and Dubois' Asynchronous Fixed-Point Theory". In: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. 2018, pp. 623–639. DOI: 10.1007/978-3-319-94821-8_37. URL: https://doi.org/10.1007/978-3-319-94821-8%5C_37.