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

April 25, 2019

THESIS PROPOSAL

-- *Supervisors*
Jacques Carette
Wolfram Kahl

-- *Emails*
carette@mcmaster.ca
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.

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

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.

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**
≈ class		≈ object
≈ type		≈ value
≈ theorem statement		≈ witnessing proof
≈ specification		≈ implementation
≈ interface		≈ implementation
≈ signature		≈ algebra
≈ logic		≈ 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 [Coq18]. Since Coq module types are essentially contexts, the module type `X <+ Y <+ Z` is really the catenation of contexts, where later items may depend on former items. The Maude [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
  infixr 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 (x ⨟_) leftId

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

Monoids as Typeclasses

```
record HasMonoid (Carrier : Set) : Set₁ where
  infixr 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 (x ⨟_) leftId

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 `∑ 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 _⨟_ = proj₁ (proj₂ (proj₂ M)))
                   → ∀ (x y : proj₁ M)  →  x ⨟ Id ⨟ y  ≡  x ⨟ y
pop-Id-Σ {{M}} x y = ≡.cong (x ⨟_) (lid {y})
                     where  _⨟_    = proj₁ (proj₂ (proj₂ M))
                            lid    = proj₁ (proj₂ (proj₂ (proj₂ M)))
```

Instances and their use are as follows.

Instance Declarations

```
instance
   ℕ-record : Monoid-Record
   ℕ-record = record { Carrier = ℕ; Id = 0; _⨟_ = _+_; ⋯ }

   ℕ-tc : HasMonoid ℕ
   ℕ-tc = record { Id = 0; _⨟_ = _+_; ⋯ }

   ℕ-Σ : Monoid-Σ
   ℕ-Σ = ℕ , 0 , _+_ , ⋯
```

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-Σₜ
```

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

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

```
Carrier_t : Monoid-Σ → Set
Carrier_t = 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
           ; leftId = lid ; rightId = rid ; assoc = assoc
           } -- Term construed by 'Agsy',
             -- the mechanical proof search.

from-record-to-usual-type : Monoid-Record → Monoid-Σ
from-record-to-usual-type M = -- syntatic data
                              Carrier M , Id M , _⨟_ M ,
                              -- semantic proofs
                              leftId M , rightId M , assoc M
  where open Monoid-Record
        -- Converting 'Monoid-Record' into a product
```

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, we position them in a `module` declaration as follows.

Agda

```
module Monoid-Telescope-User
  (Carrier : Set) (Id : Carrier) (_⨟_ : Carrier → Carrier → Carrier)
  (leftId  : ∀{x} → Id ⨟ x ≡ x) (rightId : ∀{x} → x ⨟ Id ≡ x)
  (assoc   : ∀ x y z → (x ⨟ y) ⨟ z ≡ x ⨟ (y ⨟ z))
  where

  pop-Id_m : ∀(x y : Carrier)  →  x ⨟ (Id ⨟ y)  ≡  x ⨟ y
  pop-Id_m x y = ≡.cong (x ⨟_) (leftId M {y})
```

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 ; _⨾_ = _⨾_
           ; leftId = leftId ; rightId = rightId ; 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 _⨾_ leftId rightId 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 $\mathbb{N}$-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-symmetry_m : ∀ (x y : ℕ) → x + y  ≡  y + x
ℕ-symmetry_m = symmetry_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.

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
  ::= .                          -- empty context
    | x : T [:= T],              -- context with declaration, optional definition
    | includes X,                -- theory inclusion

-- Terms
T ::= x | T1 T2 | λ x : T' • T -- variables, application, lambdas
    |  x : T' • T                -- dependent product
    | [] | ⟨⟩ | T.x             -- record "[type]" and "⟨element⟩" formers, projections
    | Mod X                      -- contravariant "theory to record" internalisation

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

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

-- Example theory hierarchy of signatures, abbrevating "( 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 `, P ⊢ t : T` then `, 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

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

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-dependent-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.

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 there may be no semantic difference between them in a dependently-typed setting, as [MRK18] present a type theoretic calculus of a variant of record types that corresponds to theories.

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 construed as syntactic sugar. 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"!

It goes without saying, these are preliminary goals, as the outcomes are likely to change and evolve multiple times as the research is carried out.

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).

[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.

[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.

[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).

[Coq18] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*. Apr. 2018. DOI: 10.5281/zenodo.1219885. URL: https://hal.inria.fr/hal-01954564.

[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.

[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.

[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.

[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.

[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.

[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.

[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).

[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.

[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.

[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.

[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).