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*	-- *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.

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 -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 [RS09], with support for them usually being conservative: Support for structuring-mechanisms elaborates, or rewrites, into the language of the ambient system's logic. Conservative extensions are reasonable to avoid bootstrapping new foundations altogether but they come at the cost of limiting expressiveness to the existing foundations; thereby possibly producing awkward or unusual uses of linguistic phrases of the ambient language.

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

[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`.

[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`.

[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`.

[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].

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

[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.

[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.

[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.

[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.

[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.

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

[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.

[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.