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

1 Introduction —The Proposal's "Story" 2

2 Current Approaches 3

3 Solution Requirements 4

4 Approach and Timeline 5

5 Conclusion 6

References 7

Chapter 1

Introduction —The Proposal's "Story"

Importance of narrative

The importance of the narrative is that it forces the remaining sections to tie into the ideas and questions posed here. Furthermore, it servers as a guide map for the remainder of the thesis proposal. Additionally, it serves as a writing prompt insisting each sentence of our proposal relates or contributes to our core mood: The disillusionment with computational organisation can be rectified by using dependent types to bring about a homogeneous treatment of structuring mechanisms.

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

[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.

[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.