Do-it-yourself Module Systems

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

Department of Computing and Software

McMaster University

Musa Al-hassy

May 26, 2020

PhD Thesis

-- *Supervisors*
Jacques Carette
Wolfram Kahl

-- *Emails*
carette@mcmaster.ca
kahl@cas.mcmaster.ca

Chapter 1

Introduction

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

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

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

2. **Practical$_2$: Unbundling** In general, in a DTL, *packages behave like functions* in that they may have a subset of their contents designated as *parameters exposed at the type-level* which users can *instantiate*. Unfortunately, library developers generally provide only a few *variations* on *a* package; such as having no parameters or having only *functional symbols* as parameters —c.f., the carrier `C` and operation $\oplus$ in figure 1. Whereas functions can *bundle-up* or *unbundle* their parameters using currying and

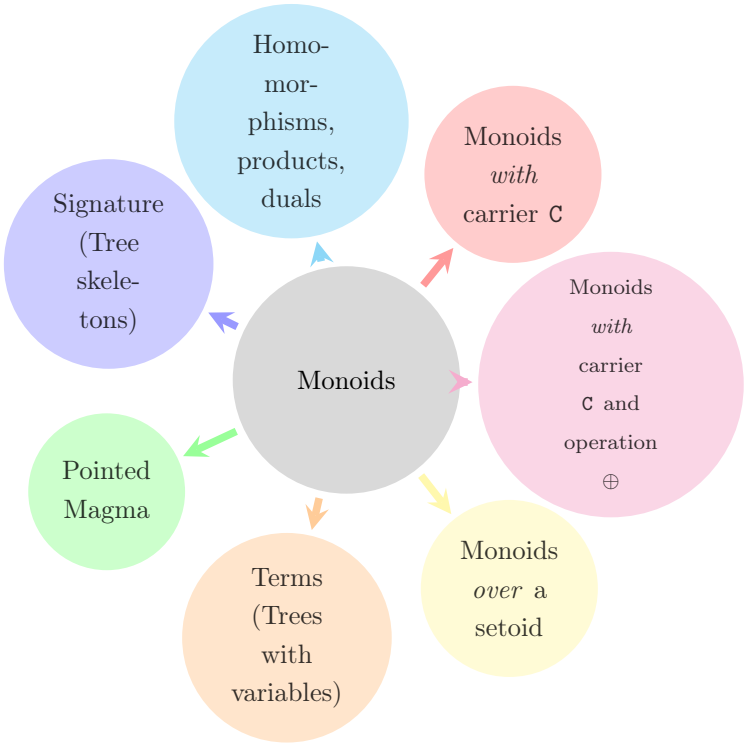

Figure 1.1: Deriving related *types* from *the* definition of monoids

uncurrying, only the latter is generally supported and, even then, not in an elegant fashion. Rather than provide *several variations* on a package, it would be more economical to provide one singular fully-bundled package and have an operator that allows users to *declaratively*, "on the fly", expose package constituents as parameters.

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

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

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

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

1.1 Thesis Overview

The remainder of the thesis is organised as follows.

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

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

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

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

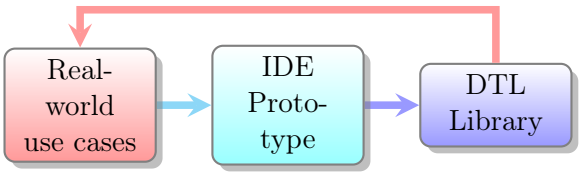

Figure 1.2: Approach for a **practical** framework

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

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

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

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

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

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

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

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

Bibliography

[Kah18] Wolfram Kahl. *Relation-Algebraic Theories in Agda*. 2018. URL: http://relmics.mcmaster.ca/RATH-Agda/ (visited on 10/12/2018) (cit. on p. 2).