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

October 30, 2020

PhD Thesis

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

Abstract

Can parameterised records and algebraic datatypes —i.e., Π-, Σ-, and $\mathcal{W}$-types— be derived from one pragmatic declaration?

Record types give a universe of discourse, parameterised record types fix parts of that universe ahead of time, and algebraic datatypes give us first-class syntax, whence evaluators and optimisers.

The answer is in the affirmative. Besides a practical shared declaration interface, which is extensible in the language, we also find that common data structures correspond to simple theories.

A middle-path with margins

Imagine having to stop reading mid-sentence, go to the bottom of the page, read a footnote, then stumble around till you get back to where you were reading$^{\alpha}$. Even worse is when one seeks a cryptic abbreviation and must decode a world-away, in the references at the end of the document.

α No more such oppresion!

I would like you to be able to read this work *smoothly, with minimal interpretations.* As such, inspired by [3] among others, we have opted to include "mathematical graffiti" in the margins. In particular, the margins side notes may have *informal and optioniated* remarks$^{\beta}$. We're trying to avoid being too dry, and aim at being somewhat light-hearted.

[3] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994. ISBN: 0-201-55802-5. URL: https://www-cs-faculty.stanford.edu/%5C%7Eknuth/gkp.html

β Professional academic writing to the left; here in the right we take a relaxed tone.

However, the cost of utilising margin space is that the overall page count may be 'over-exaggerated'$^{\gamma}$. Nonetheless, I have found long empty columns of margin space *yearning* to be filled with explanatory remarks, references, or somewhat helpful diagrams. Paraphrasing Hofstadter [4], the little pearls in the margins were so connected in my own mind with the ideas that I was writing about that for me to deprive my readers of the connection that I myself felt so strongly would be nothing less than perverse.

γ Which doesn't matter, since you're likely reading this online!

[4] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books Inc., 1979

Contents

0 Draft[0]

1 The `PackageFormer` Prototype

From the lessons learned from spelunking in a few libraries, we concluded that metaprogramming is a reasonable road on the journey toward first-class modules in DTLs. As such, we begin by forming an 'editor extension' to Agda with an eye toward a small number of 'meta-primitives'[2] for forming combinators on modules. The extension is written in Lisp, an excellent language for rapid prototyping. The purpose of writing the editor extension is not only to show that the 'flattening' of value terms and module terms is feasible[3] ; but to also show that ubiquitous packaging combinators can be generated[4] from a small number of primitives. The resulting tool resolves many of the issues discussed in section **??**.

For the interested reader, the full implementation is presented *literately* as a discussion at `https://alhassy.github.io/next-700-module-systems/prototype/package-former.html`. We will not be discussing any Lisp code in particular.

[2]Section 4.3 contains an example-driven approach

[3]Indeed, the MathScheme [1] prototype already shows this.

[4]Just as the primitive of a programming language permit arbitrarily complex programs to be written.

Chapter Contents

1.1 Why an editor extension?

The prototype[5] *rewrites* Agda phrases from an extended Agda syntax to legitimate existing syntax; it is written as an Emacs editor extension to Emacs' Agda interface, using Lisp [2]. Since Agda code is predominately written in Emacs, a practical and pragmatic editor extension would need to be in Agda's de-facto IDE[6] , Emacs. Moreover, Agda development involves the manipulation of Agda source code by Emacs Lisp —for example, for case splitting and term refinement tactics— and so it is natural to extend these ideas. Nonetheless, at a first glance, it is humorous[7] that a module extension for a statically dependently-typed language is written in a dynamically type checked language. However, *a lack of static types means some design decisions can be deferred as much as possible.*

[5]A prototype's raison d'etre is a testing ground for ideas, so its ease of development may well be more important than its usability.

[2] Paul Graham. *ANSI Common Lisp.* USA: Prentice Hall Press, 1995. ISBN: 0133708756

Why Emacs?

[6]**IDE**: Interactive Development Environment

[7]None of my colleagues thought Lisp was at all the 'right' choice; of-course, none of them had the privilege to use the language enough to appreciate it for the wonder that it is.

Unless a language provides an extension mechanism, one is forced to either alter the language's compiler or to use a preprocessing tool —both have drawbacks. The former[8] is *dangerous*; e.g., altering the grammar of a language requires non-trivial propagated changes throughout its codebase, but even worse, it could lead to existing language features to suddenly break due to incompatibility with the added features. The latter is *tiresome*[9] : It can be a nuisance to remember always invoke a preprocessor before compilation or type-checking, and it becomes extra baggage to future users of the codebase —i.e., a further addition to the toolchain that requires regular maintenance in order to be kept up to date with the core language. A middle-road between the two is not always possible.

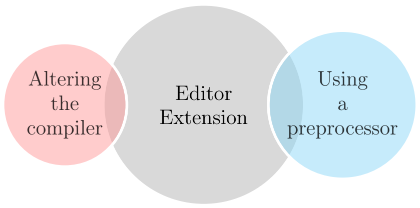
Why an editor extension? Because we quickly needed a *convenient* prototype to actually "figure out the problem".

[8]Instead of "hacking in" a new feature, one could instead carefully research, design, and implement a new feature.

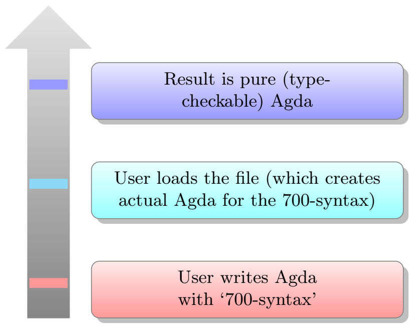
[9]Unless one uses a sufficiently flexible IDE that allows the seemless integration of preprocessing tools; which is exactly what we have done with Emacs.

However, if the language's community subscribes to *one* IDE, then a reasonable approach to extending a language would be to *plug-in* the necessary preprocessing —to transform the extended language into the pure core language— in a saliently *silent* fashion such that users need not invoke it manually.

A reasonable middle path to growing a language



Moreover, to mitigate the burden of increasing the toolchain, the salient preprocessing would *not transform user code* but instead *produce auxiliary files* containing core language code which are then *imported* by user code —furthermore, such import clauses could be automatically inserted when necessary. The benefit here is that *library users* need not know about the extended language features; since all files are in the core language with extended language feature appearing in special comments. Details can be found in section **??**.


Result is pure (type-checkable) Agda
User loads the file (which creates actual Agda for the 700-syntax)
User writes Agda with '700-syntax'

How does it work? All stages transpire in *one* user-written file

Why Lisp? Emacs is extensible using Elisp[10] wherein literally every key may be remapped and existing utilities could easily be altered *without* having to recompile Emacs. In some sense, Emacs is a Lisp interpreter and state machine. This means, we can hook our editor extension *seamlessly into the existing Agda interface* and even provide tooltips, among other features[11] , to quickly see what our extended Agda syntax transpiles into.

[10]Emacs Lisp is a combination of a large porition of Common Lisp and a editor language supporting, e.g., buffers, text elements, windows, fonts.

[11]E.g., since Emacs is a self-documenting editor, whenever a user of our tool wishes to see the documentation of a module combinator that they have written, or to read its Lisp elaboration, they merely need to invoke Emacs' help system —e.g., `C-h o` or `M-x describe-symbol`.

Finally, Lisp uses a rather small number of constructs, such as macros and lambda, which themselves are used to build 'primitives', such as `defun` for defining top-level functions [5]. Knowing this about Lisp encourages us to emulate this expressive parsimony.

[5] Doug Hoyte. *Let Over Lambda*. Lulu.com, 2008. ISBN: 1435712757

Bibliography

Here are the references in citation order.

[1] Jacques Carette et al. *The MathScheme Library: Some Preliminary Experiments*. 2011. arXiv: `1106.1862v1 [cs.MS]`.

[2] Paul Graham. *ANSI Common Lisp*. USA: Prentice Hall Press, 1995. ISBN: 0133708756.

[3] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994. ISBN: 0-201-55802-5. URL: `https://www-cs-faculty.stanford.edu/%5C%7Eknuth/gkp.html`.

[4] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books Inc., 1979.

[5] Doug Hoyte. *Let Over Lambda*. Lulu.com, 2008. ISBN: 1435712757.