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

November 2, 2020

PhD Thesis

-- Supervisors
Jacques Carette
Wolfram Kahl

-- Emails
carette@mcmaster.ca
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[α]. 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 [4] among others, we have opted to include "mathematical graffiti" in the margins. In particular, the margins side notes may have *informal and optioniated* remarks[β]. We're trying to avoid being too dry, and aim at being somewhat light-hearted.

[4] 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.

Dijkstra [2] might construe the graffiti as *mathematical politeness* that could potentially save the reader a minute. Even though a characteristic of academic writing is its terseness[ω], we don't want to baffle or puzzle our readers, and so we use the informality of the graffiti to say what we mean bluntly, *but* it may be less accurate or not as formally justifiable as the text proper.

[2] Edsger W. Dijkstra. *The notational conventions I adopted, and why.* circulated privately. July 2000. URL: http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF

ω "It's so obvious, I won't waste time on it"; i.e., "It's an exercise to the reader to figure out what I'm really saying." Elaboration removes mystery and some authors might prefer academia be exclusive.

> *Some consider the puzzles that are created by their omissions as spicy challenges, without which their texts would be boring; others shun clarity lest their worth is considered trivial. [...] Some authors believe that, in order to keep the reader awake, one has to tickle him with surprises. [...] essential for earning the respect of their readership.*
> *—Edsger Dijkstra [2]*

[2] Edsger W. Dijkstra. *The notational conventions I adopted, and why.* circulated privately. July 2000. URL: http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF

A superficial cost of utilising margin space is that the overall page count may be 'over-exaggerated'[γ]. Nonetheless, I have found long empty columns of margin space *yearning* to be filled with explanatory remarks, references, or somewhat helpful diagrams. Paraphrasing Hofstadter [5], 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!

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

Contents

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 [3]. 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.

[3] 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.

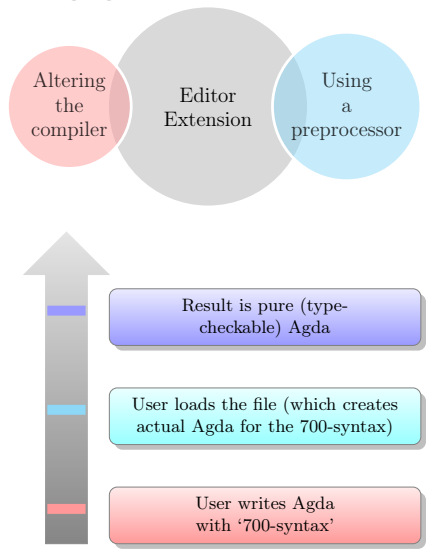
Why an editor extension? Because we quickly needed a *convenient* prototype to actually "figure out the problem".

[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 1.2.

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 [6]. Knowing this about Lisp encourages us to emulate this expressive parsimony.

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

1.2 Aim: *Scrap the Repetition*

Programming Language research is summarised, in essence, by the question: *If $\mathcal{X}$ is written manually, what information $\mathcal{Y}$ can be derived for free?* Perhaps the most popular instance is *type inference*: From the syntactic structure of an expression, its type can be derived.

From a context, the PackageFormer editor extension can generate the many common design patterns discussed earlier in section **??**; such as unbundled variations of any number wherein fields are exposed as parameters at the type level, term types for syntactic manipulation, arbitrary renaming, extracting signatures, and forming homomorphism types. In this section we discuss how PackageFormer works and provide a 'real-world' use case, along with a discussion.

With the extension, Agda's usual `C-c C-l` command parses special comments containing fictitious Agda declarations, produces an auxiliary Agda file which it ensures is imported in the current file, then control is passed to the usual Agda typechecking mechanism.

Below is example code that can occur in the specially recognised comments. The first eight lines, starting at line 1, are essentially an Agda record declaration but the field qualifier is absent. The declaration is intended to name an abstract context, a sequence of "name : type" pairs as discussed at length in chapter **??**, but we use the name PackageFormer instead of 'context, signature, telescope', nor 'theory' since those names have existing biased connotations — besides, the new name is more 'programmer friendly'.

In the code block, the names have been chosen to stay relatively close to the real-world examples presented in chapter **??**. The name **M-Set** comes from *monoid acting on a set*; in our example, Scalar values may act on Vector values to produce new Scalar values. The programmer may very well appreciate this example if the names Scalar, 𝟙, _×_, Vector, _·_ were chosen to be Program, do-nothing, _⨟_, Input, run. With this new naming, leftId says *running the empty program on any input, leaves the input unchanged*, whereas assoc says *to run a sequence of programs on an input, the input must be threaded through the programs*. Whence, **M-Sets abstract program execution**.

M-Sets are sets 'Scalar' acting '_·_' on semigroups 'Vector'

```
1  PackageFormer M-Set : Set₁ where
2    Scalar  : Set
3    Vector  : Set
4    _·_     : Scalar → Vector → Vector
5    𝟙       : Scalar
6    _×_     : Scalar → Scalar → Scalar
7    leftId  : {v : Vector}  →  𝟙 · v  ≡  v
8    assoc   : {a b : Scalar} {v : Vector} →  (a × b) · v
9                                          ≡  a · (b · v)
```

Different Ways to Organise ("interpret" / "use") M-Sets

Now to actually use this context ...

```
9  Semantics  = M-Set -⊕- record
10 Semantics𝒟 = Semantics -⊕- rename (λ x → (concat x "𝒟"))
11 Semantics₃ =  Semantics :waist 3
12
13 Left-M-Set  = M-Set -⊕- record
14 Right-M-Set = Left-M-Set -⊕- flipping "_·_" :renaming "leftId
   ↪  to rightId"
15
16 ScalarSyntax = M-Set -⊕- primed -⊕- data "Scalar′"
17 Signature    = M-Set -⊕- record -⊕- signature
18 Sorts        = M-Set -⊕- record -⊕- sorts
19
20 𝒱-one-carrier   = renaming "Scalar to Carrier; Vector to
   ↪  Carrier"
21 𝒱-compositional = renaming "_×_ to _⨟_; _·_ to _⨟_"
22 𝒱-monoidal      = one-carrier -⊕- compositional -⊕- record
23
24 LeftUnitalSemigroup = M-Set -⊕- monoidal
25 Semigroup           = M-Set -⊕- keeping "assoc" -⊕- monoidal
26 Magma               = M-Set -⊕- keeping "_×_" -⊕- monoidal
```

M-Sets as records, possibly with renaming or parameters.

⋆ ⋆ ⋆

Duality; we might want to change the order of the action, say, to write `evalAt x f` instead of `run f x` —using the program-input interpretation of M-Sets above.

⋆ ⋆ ⋆

Keeping only the 'syntactic interface', say, for serialisation or automation.

⋆ ⋆ ⋆

Collapsing different features to obtain the notion of "monoid".

⋆ ⋆ ⋆

Obtaining parts of the monoid hierarchy (see chapter 3) from M-Sets

These manually written ∼25 lines elaborate into the ∼100 lines of

raw, legitimate, Agda syntax below —line breaks are denoted by the symbol '↪' rather than inserted manually, since all subsequent code snippets in this section are **entirely generated** by PackageFormer. The result is nearly a **400% increase in size**; that is, our fictitious code will save us a lot of repetition.

Let's discuss what's actually going on here.

The first line declares the context of M-Sets using traditional Agda syntax " record M-Set : Set₁ where " except the we use the word PackageFormer to avoid confusion with the existing record concept, but[12] we also *omit* the need for a field keyword and *forbid* the existence of parameters. Such abstract contexts have no concrete form in Agda and so no code is generated; the second snippet above[13] shows sample declarations that result in legitimate Agda.

PackageFormer module combinators are called *variationals* since they provide a variation on an existing grouping mechanism. The syntax $\texttt{p} \oplus\!\!\rightarrow v_1 \oplus\!\!\rightarrow \cdots \oplus\!\!\rightarrow v_n$ is tantamount to explicit forward function application $v_n\ (v_{n-1}\ (\cdots\ (v_1\ \texttt{p})))$. With this understanding, we can explain the different ways to organise M-sets.

The next page shows, first, the elaboration of lines 9-11 and, in the second snippet, the elaborations of lines 12-14.

In line 9, the record variational is invoked to transform the abstract context M-Set into a valid Agda record declaration, with the key word field inserted as necessary. Later, its first 3 fields are lifted as parameters using the meta-primitive :waist.

The waist is the number of parameters exposed; recall $\Pi^w\Sigma$ from chapter 2.

Arbitrary functions act on modules

When only one variational is applied to a context, the one and only '$\oplus\!\!\rightarrow$' may be omitted. As such, $\texttt{Semantics}_3$ is defined as Semantics rename f, where f is the decoration function. In this form, one is tempted to believe

```
_rename_ : PackageFormer → (Name → Name) → PackageFormer
```

That is, we have a binary operation in which functions may act on modules —this is yet a new feature that Agda cannot perform.

Notice how SemanticsD was *built from* a concrete context, namely the Semantics record. As such, every instance of SemanticsD can be transformed as an instance of Semantics: This view[14] —see Section **??**— is automatically generated and named toSemantics above, by default. Likewise, Right-M-Set was derived from Left-M-Set and so we have automatically have a view Right-M-Set → Left-M-Set.

[12] **Conflating fields, parameters, and definitional extensions:** The lack of a field keyword and forbidding parameters means that arbitrary programs may 'live within' a PackageFormer and it is up to a variational to decide how to treat them and their optional definitions.

[13] For every (special comment) declaration $\mathcal{L} = \mathcal{R}$ in the source file, the name $\mathcal{L}$ obtains a tooltip which mentions its specification $\mathcal{R}$ and the resulting legitimate Agda code. This feature is indispensable as it lets one generate grouping mechanisms and quickly ensure that they are what one intends them to be.

[14] It is important to remark that the mechanical construction of such views (coercions) is **not built-in**, but rather a *user-defined* variational that is constructed from PackageFormer's meta-primitives.

Lines 9-11 — Record / decorated renaming / typeclass forms

```
{- Semantics           = M-Set -⊕> record -}
record Semantics : Set₁ where
    field Scalar                : Set
    field Vector                : Set
    field _·_           : Scalar → Vector → Vector
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar
    field leftId                : {v : Vector}  →  𝟙 · v  ≡  v
    field assoc         : {a b : Scalar} {v : Vector} → (a × b) · v  ≡  a · (b · v)


{- Semantics𝒟          = Semantics -⊕> rename (λ x → (concat x "𝒟")) -}
record Semantics𝒟 : Set₁ where
    field Scalar𝒟               : Set
    field Vector𝒟               : Set
    field _·𝒟_          : Scalar𝒟 → Vector𝒟 → Vector𝒟
    field 𝟙𝒟            : Scalar𝒟
    field _×𝒟_          : Scalar𝒟 → Scalar𝒟 → Scalar𝒟
    field leftId𝒟               : {v : Vector𝒟}  →  𝟙𝒟 ·𝒟 v  ≡  v
    field assoc𝒟                : {a b : Scalar𝒟} {v : Vector𝒟} → (a ×𝒟 b) ·𝒟 v  ≡  a ·𝒟
    ↪  (b ·𝒟 v)
    toSemantics         : let View X = X in View Semantics ;    toSemantics = record {Scalar =
    ↪  Scalar𝒟;Vector = Vector𝒟;_·_ = _·𝒟_;𝟙 = 𝟙𝒟;_×_ = _×𝒟_;leftId = leftId𝒟;assoc =
    ↪  assoc𝒟}


{- Semantics₃          =  Semantics :waist 3 -}
record Semantics₃ (Scalar : Set) (Vector : Set) (_·_ : Scalar → Vector → Vector) : Set₁ where
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar
    field leftId                : {v : Vector}  →  𝟙 · v  ≡  v
    field assoc         : {a b : Scalar} {v : Vector} → (a × b) · v  ≡  a · (b · v)
```

Lines 13-14 — Duality: Sets can act on semigroups from the left or the right

```
record Left-M-Set : Set₁ where
    field Scalar                : Set
    field Vector                : Set
    field _·_           : Scalar → Vector → Vector
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar
    field leftId                : {v : Vector}  →  𝟙 · v  ≡  v
    field assoc         : {a b : Scalar} {v : Vector} → (a × b) · v  ≡  a · (b · v)


{- Right-M-Set         = Left-M-Set -⊕> flipping "_·_" :renaming "leftId to rightId" -}
record Right-M-Set : Set₁ where
    field Scalar                : Set
    field Vector                : Set
    field _·_           :  Vector  → Scalar  →  Vector
    field 𝟙             : Scalar
    field _×_           : Scalar → Scalar → Scalar
    field rightId               : let _·_ = λ x y → _·_ y x in {v : Vector}  →  𝟙 · v  ≡  v
    field assoc         : let _·_ = λ x y → _·_ y x in {a b : Scalar} {v : Vector} → (a × b)
    ↪  · v  ≡  a · (b · v)
    toLeft-M-Set                : let _·_ = λ x y → _·_ y x in let View X = X in View
    ↪  Left-M-Set ;      toLeft-M-Set  = let _·_ = λ x y → _·_ y x in   record {Scalar =
    ↪  Scalar;Vector = Vector;_·_ = _·_;𝟙 = 𝟙;_×_ = _×_;leftId = rightId;assoc = assoc}
```

Likewise, line 13, mentions another combinator

```
_flipping_ : PackageFormer → Name → PackageFormer
```

This combinator also takes an *optional keyword argument* `:renaming`, which simply renames the given pair. The notation of keyword arguments is inherited[15] from Lisp.

[15] More accurately, the '⊕'-based mini-language for variationals is realised as a Lisp macro and so, in general, the right side of a declaration in 700-comments is interpreted as valid Lisp modulo this mini-language: `PackageFormer` names and variationals are variables in the Emacs environment —for declaration purposes, and to avoid touching Emacs specific utilities, variationals `f` are actually named `𝒱-f`. One may quickly obtain the documentation of a variational `f` with `C-h o RET 𝒱-f` to see how it works.

⋆ ⋆ ⋆

An algebraic data type is a tagged union of symbols, terms, and so is one type —see section ??.

Recall from Chapter ??, symbols that target `Set` are considered sorts and if we keep only the symbols targeting a sort, we have a signature. By allowing symbols to be of type `Set`, we actually have **generalised contexts**.

The priming decoration in `ScalarSyntax` is needed so that the names `𝟙, _×_` do not pollute the global name space.

Next, in line 16, we view a context as such a termtype by declaring one sort of the context to act as the termtype (carrier) and then keep only the function symbols that target it —this is the **core idea** that is used when we operate on Agda `Term`s in the next chapter.

Lines 16-18 Termtypes and lawless presentations

```
data ScalarSyntax : Set where
    𝟙′          : ScalarSyntax
    _×′_        : ScalarSyntax → ScalarSyntax →
    ↪  ScalarSyntax


{- Signature          = M-Set -⊕- record -⊕- signature -}
record Signature : Set₁ where
    field Scalar              : Set
    field Vector              : Set
    field _·_                 : Scalar → Vector → Vector
    field 𝟙                   : Scalar
    field _×_                 : Scalar → Scalar → Scalar


{- Sorts              = M-Set -⊕- record -⊕- sorts -}
record Sorts : Set₁ where
    field Scalar              : Set
    field Vector              : Set
```

Finally, starting with line 20, declarations start with " `𝒱-` " to indicate that a new variation *combinator* is to be formed, rather than a new *grouping* mechanism. For instance, the user-defined `one-carrier` variational identifies both the `Scalar` and `Vector` sorts, whereas `compositional` identifies the binary operations; then, finally, `monoidal` performs both of those operations and also produces a concrete Agda `record` formulation. Below, in the final code snippet of this section, are the elaborations of using these new new user-defined variationals.

User defined variationals are applied as if they were built-ins.

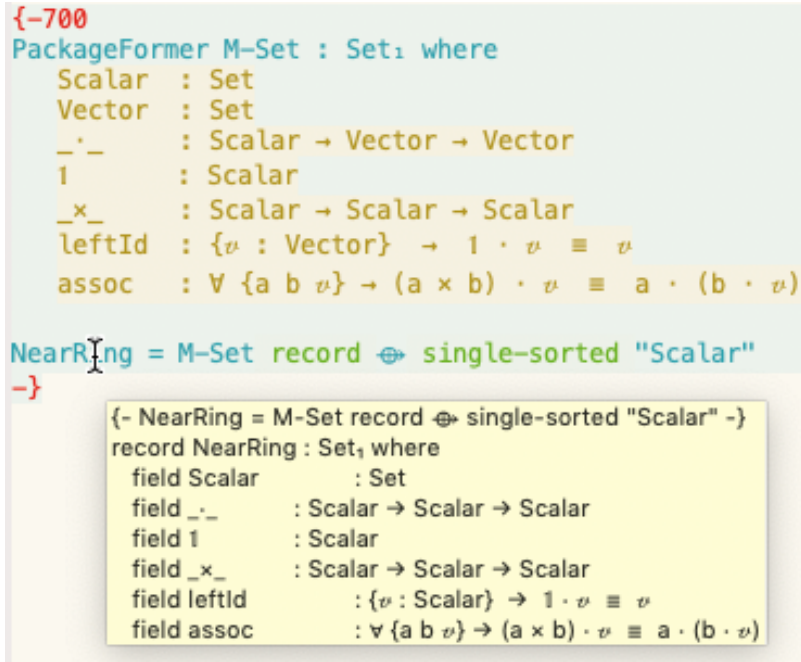
Lines 24-26 — Conflating features gives familiar structures

```
record LeftUnitalSemigroup : Set₁ where
    field Carrier               : Set
    field _⨟_           : Carrier → Carrier → Carrier
    field 𝟙             : Carrier
    field leftId                : {v : Carrier}  →  𝟙 ⨟ v  ≡  v
    field assoc         : {a b : Carrier} {v : Carrier} → (a ⨟ b) ⨟ v  ≡  a ⨟ (b ⨟ v)


{- Semigroup           = M-Set ⟴ keeping "assoc" ⟴ monoidal -}
record Semigroup : Set₁ where
    field Carrier               : Set
    field _⨟_           : Carrier → Carrier → Carrier
    field assoc         : {a b : Carrier} {v : Carrier} → (a ⨟ b) ⨟ v  ≡  a ⨟ (b ⨟ v)


{- Magma               = M-Set ⟴ keeping "_×_" ⟴ monoidal -}
record Magma : Set₁ where
    field Carrier               : Set
    field _⨟_           : Carrier → Carrier → Carrier
```

As shown in the figure below, the source file is furnished with tooltips displaying the special comment that a name is associated with, as well as the full elaboration into legitimate Agda syntax. In addition, the above generated elaborations also document the special comment that produced them. Moreover, since the editor extension results in valid code in an auxiliary file, future users of a library need not use the PackageFormer extension at all —thus we essentially have a static **editor tactic** similar to Agda's (Emacs interface) proof finder.



Hovering to show details. Notice special syntax has default colouring: Red for PackageFormer delimiters, yellow for elements, and green for variationals.

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] Edsger W. Dijkstra. *The notational conventions I adopted, and why*. circulated privately. July 2000. URL: http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF.

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

[4] 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.

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

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