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

August 21, 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.

Contents

Chapter 1

Introduction

unchanged

Chapter 2

Packages and Their Parts

unchanged

Prerequisite of the reader: Going forward, it is assumed that the reader is comfortable programming with Haskell, and the associated menagerie of Category Theory concepts that are usually present in the guise of Functional Programming. In particular, this includes ‘practical’ notions such as typeclasses and instance search, as well as ‘theoretical’ notions such categorial limits and colimits, lattices —a kind of category with products— and monoids —possibly in arbitrary monoidal categories, as is the case with monads. Moreover, we assume the reader to have **actually** worked with a dependently-typed language; otherwise, it *may* be difficult to appreciate the solutions to the problems addressed in this thesis —since they could not be expressed in languages without dependent-types and are thus ‘not problems’.

Chapter 3

Motivating the problem —Examples from the Wild

Tedium is for machines; interesting problems are for people.

In this section, we showcase a number of problems that occur in developing libraries of code *within* dependently-typed languages. We will refer back to these real-world examples later on when developing our frameworks for reducing their tedium and size. The examples are extracted from Agda libraries focused on mathematical domains, such as algebra and category theory. It is not important to understand the application domains, but how modules are organised and used. The examples will focus on readability (sections 3.1, 3.2) and on mixing-in features to an existing module (sections 3.3, 3.4, 3.5). In order to make the core concepts acceptable, we will occasionally render examples using the simple algebraic structures: Magma , Semigroup, and Monoid [1].

Incidentally, the common solutions to the problems presented may be construed as **design patterns for dependently-typed programming**. Design patterns are algorithms yearning to be formalised. The power of the host language dictates whether design patterns remain as informal directions to be implemented in an ad-hoc basis then checked by other humans, or as a library methods that are written once and may be freely applied by users. For instance, the Agda `Algebra.Morphism` "library"[2] presents *only* an example(!) of the homomorphism design pattern —which shows how to form operation-preserving functions for algebraic structures. The documentation reads: `An example showing how a morphism type can be defined`. An example, rather than a library method, is all that can be done

[1] A *magma* (`C, ⨾`) is a set `C` and a binary operation `_⨾_ : C → C → C` on it; a *semigroup* is a magma whose operation is associative, `∀ x, y, z • (x ⨾ y) ⨾ z = x ⨾ (y ⨾ z)`; and a *monoid* is a semigroup that has a point `Id : C` acting as the identity of the binary operation: `∀ x • x ⨾ Id = x = Id ⨾ x`.

[2] All references to the Agda Standard Library refer to version 0.7. The current version is 1.3, however, for the `Algebra.Morphism` library, the newer library only refactors the one monolithic homomorphism example into a fine grained hierarchy of homomorphisms. The library can be accessed at `https://github.com/agda/agda-stdlib`.

since the current implementation of Agda does not have the necessary meta-programming utilities to construct new types in a practical way —at least, not out of the box.

3.1 Simplifying Programs by Exposing Invariants at the Type Level

In theory, lists and vectors are the same —where the latter are essentially lists indexed by their lengths. In practice, however, the additional length information stated up-front as an integral part of the data structure makes it not only easier to write programs that would otherwise by awkward or impossible in the latter case. For instance, below we demonstrate that the function `head`, which extracts the first element of a non-empty list, not only has a difficult type to read, but also requires an auxiliary relation in order to be expressed. In contrast, the vector variant has a much simpler type with the non-emptiness proviso expressed by requesting a positive length.

Exposing Information At the Type Level

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A

data Vec (A : Set) : ℕ → Set where
  []   : Vec A 0
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)

data not-null {A : Set} : List A → Set where
  indeed : ∀ {x xs} → not-null (x :: xs)

head : ∀ {A} → Σ xs : List A • not-null xs → A
head ([] , ())
head (x :: xs , indeed) = x

head' : ∀ {A n} → Vec A (suc n) → A
head' (x :: xs) = x
```

In the definition of `head`, we pattern match on the possible ways to form a list —namely, `[]` and `_::_`. In the first case, we perform *case analysis* on the shape of the proof of `not-null []`, but there is no way to form such a proof and so we have "defined" the first clause of `head` using *a definition by zero-cases* on the `non-null` proof. The 'absurd pattern' `()` indicates the impossibility of a construction.

This phenomenon applies not only to derived concepts such as non-emptiness, but also to explicit features of a datatype. A common scenario is when two instances of an algebraic structure share the same carrier and thus it is reasonable to connect the two somehow by a coherence axiom. Perhaps the most popular instance of this scenario is in the setting of rings: There is an additive monoid `(R, +, 1)` and a multiplicative monoid `(R, ×, 0)` on the same underlying set `R`, and their interaction is dictated by two distributivity axioms, such as $a \times (b + c) = (a \times b) + (a \times c)$. As with **`head`** above, depending on which features of a monoid are exposed upfront, such axioms may be either difficult to express or relatively easy.

For brevity, since our interest is in expressing the aforementioned distributivity axiom, we shall ignore all other features of a monoid, to obtain a magma.

Distributivity is Difficult to Express

```
record Magma₀ : Set₁ where
  field
    Carrier : Set
    _⨾_      : Carrier → Carrier → Carrier

record Distributivity₀ (Additive Multiplicative : Magma₀) : Set₁ where

  open Magma₀ Additive       renaming (Carrier to R₊; _⨾_ to _+_)
  open Magma₀ Multiplicative renaming (Carrier to R×; _⨾_ to _×_)

  field shared-carrier :  R₊ ≡ R×

  coe× : R₊ → R×
  coe× = subst id shared-carrier

  coe₊ : R× → R₊
  coe₊ = subst id (sym shared-carrier)

  field distribute₀ : ∀ {a : R×} {b c : R₊}
                    →   a × coe× (b + c)
                      ≡ coe× (coe₊(a × coe× b) + coe₊(a × coe× c))
```

It is a bit of a challenge to understand the type of `distribute₀`. Even though the carriers of the monoids are propositionally equal, `R₊ ≡ R×`, they are not the same by definition —the notion of equality wass defined in section **??**. As such, we are forced to "coe"rce back and forth; leaving the distributivity axiom as an exotic property of addition, multiplication, and coercions. Even worse, without the cleverness of declaring two coercion helpers, the typing of `distribute₀` would have been so large and confusing that the concept would be rendered near useless.

In **theory**, parameterised structures are no different from their unparameterised, or "bundled", counterparts. However, in **practice**, this is wholly untrue: Below we can phrase the distributivity axiom nearly as it was stated informally earlier since the shared carrier is declared upfront.

Distributivity is Expressed Easily with Unbundled Structures

```
record Magma₁ (Carrier : Set) : Set₁ where
  field
    _⨾_      : Carrier → Carrier → Carrier

record Distributivity₁
    (R : Set) {- The shared carrier -}
    (Additive Multiplicative : Magma₁ R)  : Set₁ where

  open Magma₁ Additive       renaming (_⨾_ to _+_)
  open Magma₁ Multiplicative renaming (_⨾_ to _×_)

  field distribute₁ : ∀ {a b c : R} →  a × (b + c) ≡ (a × b) + (a × c)
```

In contrast to the bundled definition of magmas, this form requires no cleverness to form coercion helpers, and is closer to the informal and usual distributivity statement.

By the same arguments above, the simple statement relating the two units of a ring $1 \times r + 0 = r$ —or any units of monoids sharing the same carrier— is easily phrased using an unbundled presentation and would require coercions otherwise. We invite the reader to pause at this moment to appreciate the difficulty in simply expressing this property.

Unbundling Design Pattern

If a feature of a class is shared among instances, then use an unbundled form of the class to avoid "coercion hell".

Observe that we assigned superficial renamings, aliases, to the prototypical binary operation `_⨾_` so that we may phrase the distributivity axiom in its expected notational form. This leads us to our next topic of discussion.

3.2 Renaming

The use of an idea is generally accompanied with particular notation that is accepted by its primary community. Even though the choice of bound names it theoretically irrelevant, certain communities would consider it unacceptable to deviate from convention. Here are a few examples:

$x(f)$ Using x as a *function* and f as an *argument*.; likewise $\frac{\partial x}{\partial f}$.

With the exception of discussions involving the Yoneda Lemma, or continuations, such a notation is simply *'wrong'*.

$a \times a = a$ An idempotent operation denoted by multiplication; likewise for commutative operations. It is more common to use addition or join, '$\sqcup$'.

$0 \times a \approx a$ The identity of "multiplicative symbols" should never resemble '0'; instead it should resemble '1' or, at least, 'e' —the standard abbreviation of the influential algebraic works of German authors who used "Einheit" which means "identity".

$f + g$ Even if monoids are defined with the prototypical binary operation denoted '+', it would be *'wrong'* to continue using it to denote functional composition. One would need to introduce the new name '$\circ$' or, at least, '$\cdot$'.

From the few examples above, it is immediate that to even present a prototypical notation for an idea, one immediately needs auxiliary notation when specialising to a particular instance. For example, to use 'additive symbols' such as $+, \sqcup, \oplus$ to denote an arbitrary binary operation leads to trouble in the function composition instance above, whereas using 'multiplicative symbols' such as $\times, , *$ leads to trouble in the idempotent case above. Regardless of prototypical choices, there will always be a need to rename.

Renaming Design Pattern

Use superficial aliases to better communicate an idea; especially so, when the topic domain is specialised.

Let's now turn to examples of renaming from three libraries:

1. Agda's "standard library" [20],
2. The "RATH-Agda" library [Kah18], and
3. A recent "agda-categories" library [Jac20].

Each will provide a workaround to the problem of renaming. In particular, the solutions are, respectively:

1. **Rename as needed.**
   - ◇ There is no systematic approach to account for the many common renamings.
   - ◇ Users are encouraged to do the same, since the standard library does it this way.
2. **Pack-up the *common* renamings as modules, and invoke them when needed.**
   - ◇ Which renamings are provided is left at the discretion of the designer —even 'expected' renamings may not be there since, say, there are too many choices or insufficient man power to produce them.

- ◇ The pattern to pack-up renamings leads nicely to consistent naming.

3. **Names don't matter.**

   - ◇ Users of the library need to be intimately connected with the Agda definitions and domain to use the library.
   - ◇ Consequently, there are many inconsistencies in naming.

The `open` ⋯ `public` ⋯ `renaming` ⋯ pattern shown below will be presented later, section **??**, as a library method.

3.2.1 Renaming Problems from Agda's Standard Library

Here are four excerpts from Agda's standard library, notice how the prototypical notation for monoids is renamed **repeatedly** *as needed*. Sometimes it is relabelled with additive symbols, other times with multiplicative symbols. The content itself is not important, instead the focus is on the renaming that takes place —as such, the fontsize is intentionally tiny.

Additive Renaming —IsNearSemiring

```
record IsNearSemiring {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                      (+ * : Op₂ A) (0# : A) : Set (a ⊔ ℓ)
 ↪    where
  open FunctionProperties ≈
  field
    +-isMonoid    : IsMonoid ≈ + 0#
    *-isSemigroup : IsSemigroup ≈ *
    distribʳ      : * DistributesOverʳ +
    zeroˡ         : LeftZero 0# *

  open IsMonoid +-isMonoid public
         renaming ( assoc       to +-assoc
                  ; ·-cong      to +-cong
                  ; isSemigroup to +-isSemigroup
                  ; identity    to +-identity
                  )

  open IsSemigroup *-isSemigroup public
         using ()
         renaming ( assoc    to *-assoc
                  ; ·-cong   to *-cong
                  )
```

Additive Renaming Again —IsSemiringWithoutOne

```
record IsSemiringWithoutOne {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                            (+ * : Op₂ A) (0# : A) : Set (a ⊔
 ↪    ℓ)
 where
  open FunctionProperties ≈
  field
    +-isCommutativeMonoid : IsCommutativeMonoid ≈ + 0#
    *-isSemigroup         : IsSemigroup ≈ *
    distrib               : * DistributesOver +
    zero                  : Zero 0# *

  open IsCommutativeMonoid +-isCommutativeMonoid public
         hiding (identityˡ)
         renaming ( assoc       to +-assoc
                  ; ·-cong      to +-cong
                  ; isSemigroup to +-isSemigroup
                  ; identity    to +-identity
                  ; isMonoid    to +-isMonoid
                  ; comm        to +-comm
                  )

  open IsSemigroup *-isSemigroup public
         using ()
         renaming ( assoc       to *-assoc
                  ; ·-cong      to *-cong
                  )
```

Additive Renaming a 3rd Time and Multiplicative Renaming —IsSemiringWithoutAnnihilatingZero

```
record IsSemiringWithoutAnnihilatingZero
        {a ℓ} {A : Set a} (≈ : Rel A ℓ)
        (+ * : Op₂ A) (0# 1# : A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    +-isCommutativeMonoid : IsCommutativeMonoid ≈ + 0#
    *-isMonoid            : IsMonoid ≈ * 1#
    distrib               : * DistributesOver +

  open IsCommutativeMonoid +-isCommutativeMonoid public
        hiding (identityˡ)
        renaming ( assoc       to +-assoc
                 ; ·-cong      to +-cong
                 ; isSemigroup to +-isSemigroup
                 ; identity    to +-identity
                 ; isMonoid    to +-isMonoid
                 ; comm        to +-comm
                 )

  open IsMonoid *-isMonoid public
        using ()
        renaming ( assoc       to *-assoc
                 ; ·-cong      to *-cong
                 ; isSemigroup to *-isSemigroup
                 ; identity    to *-identity
                 )
```

Additive Renaming a 4th Time and Second Multiplicative Renaming —IsRing

```
record IsRing
        {a ℓ} {A : Set a} (≈ : Rel A ℓ)
        (_+_ _*_ : Op₂ A) (-_ : Op₁ A) (0# 1# : A) : Set (a
 ↪    ⊔ ℓ)
 where
  open FunctionProperties ≈
  field
    +-isAbelianGroup : IsAbelianGroup ≈ _+_ 0# -_
    *-isMonoid       : IsMonoid ≈ _*_ 1#
    distrib          : _*_ DistributesOver _+_

  open IsAbelianGroup +-isAbelianGroup public
        renaming ( assoc               to +-assoc
                 ; ·-cong              to +-cong
                 ; isSemigroup         to +-isSemigroup
                 ; identity            to +-identity
                 ; isMonoid            to +-isMonoid
                 ; inverse             to -CONVERSEinverse
                 ; ⁻¹-cong              to -CONVERSEcong
                 ; isGroup             to +-isGroup
                 ; comm                to +-comm
                 ; isCommutativeMonoid to
 ↪    +-isCommutativeMonoid
                 )

  open IsMonoid *-isMonoid public
        using ()
        renaming ( assoc       to *-assoc
                 ; ·-cong      to *-cong
                 ; isSemigroup to *-isSemigroup
                 ; identity    to *-identity
                 )
```

At first glance, one solution would be to package up these renamings into helper modules. For example, consider the setting of monoids.

Original —Prototypical— Notations

```
record IsMonoid {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                (· : Op₂ A) (ε : A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    isSemigroup : IsSemigroup ≈ ·
    identity    : Identity ε ·

record IsCommutativeMonoid {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                           (_·_ : Op₂ A) (ε : A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    isSemigroup : IsSemigroup ≈ _·_
    identityˡ   : LeftIdentity ε _·_
    comm        : Commutative _·_

    ⋮
  isMonoid : IsMonoid ≈ _·_ ε
  isMonoid = record { ⋯ }
```

Renaming Helper Modules

```
module AdditiveIsMonoid {a ℓ} {A : Set a} {≈ : Rel A ℓ}
               {_·_ : Op₂ A} {ε : A} (+-isMonoid : IsMonoid ≈ _·_ ε)  where

   open IsMonoid +-isMonoid public
         renaming ( assoc       to +-assoc
                  ; ·-cong      to +-cong
                  ; isSemigroup to +-isSemigroup
                  ; identity    to +-identity
                  )

module AdditiveIsCommutativeMonoid {a ℓ} {A : Set a} {≈ : Rel A ℓ}
               {_·_ : Op₂ A} {ε : A} (+-isCommutativeMonoid : IsMonoid ≈ _·_ ε)
 ↪  where

   open AdditiveIsMonoid (CommutativeMonoid.isMonoid +-isCommutativeMonoid) public
   open IsCommutativeMonoid +-isCommutativeMonoid public using ()
      renaming ( comm to +-comm
               ; isMonoid to +-isMonoid)
```

However, one then needs to make similar modules for *additive notation* for `IsAbelianGroup`, `IsRing`, `IsCommutativeRing`, …. Moreover, this still invites repetition: Additional notations, as used in `IsSemiring`, would require additional helper modules.

More Necessary Renaming Helper Modules

```
module MultiplicativeIsMonoid {a ℓ} {A : Set a} {≈ : Rel A ℓ}
               {_·_ : Op₂ A} {ε : A} (*-isMonoid : IsMonoid ≈ _·_ ε)  where

   open IsMonoid *-isMonoid public
         renaming ( assoc       to *-assoc
                  ; ·-cong      to *-cong
                  ; isSemigroup to *-isSemigroup
                  ; identity    to *-identity
                  )
```

Unless carefully organised, such notational modules would bloat the standard library, resulting in difficulty when navigating the library. As it stands however, the new algebraic structures appear large and complex due to the “renaming hell” encountered to provide the expected conventional notation.

3.2.2 Renaming Problems from the RATH-Agda Library

The impressive Relational Algebraic Theories in Agda library takes a disciplined approach: Copy-paste notational modules, possibly using a find-replace mechanism to vary the notation. The use of a find-replace mechanism leads to consistent naming across different notations.

Quoting the library, *For contexts where calculation in different setoids is necessary, we provide "decorated" versions of the* ***`Setoid'`*** *and* ***`SetoidCalc`*** *interfaces:*

Seotoid$\mathcal{D}$ Renamings —$\mathcal{D}$decorated Synonyms

```
module SetoidA {i j : Level} (S : Setoid i j) = Setoid' S renaming
    ( ℓ to ℓA ; Carrier to A₀ ; _≈_ to _≈A_ ; ≈-isEquivalence to ≈A-isEquivalence
    ; ≈-isPreorder to ≈A-isPreorder ; ≈-preorder to ≈A-preorder
    ; ≈-indexedSetoid to ≈A-indexedSetoid
    ; ≈-refl to ≈A-refl ; ≈-reflexive to ≈A-reflexive ; ≈-sym to ≈A-sym
    ; ≈-trans to ≈A-trans ; ≈-trans₁ to ≈A-trans₁ ; ≈-trans₂ to ≈A-trans₂
    ; _⟨≈≈⟩_ to _⟨≈A≈⟩_ ; _⟨≈≈˘⟩_ to _⟨≈A≈˘⟩_ ; _⟨≈˘≈⟩_ to _⟨≈A˘≈⟩_
    ; _⟨≈˘≈˘⟩_ to _⟨≈A˘≈˘⟩_; _⟨≡≈⟩_ to _⟨≡≈A⟩_ ; _⟨≡≈˘⟩_ to _⟨≡≈A˘⟩_
    ; _⟨≡˘≈⟩_ to _⟨≡˘≈A⟩_ ; _⟨≡˘≈˘⟩_ to _⟨≡˘≈A˘⟩_ ; _⟨≈≡⟩_ to _⟨≈A≡⟩_
    ; _⟨≈≡˘⟩_ to _⟨≈A≡˘⟩_ ; _⟨≈˘≡⟩_ to _⟨≈A˘≡⟩_ ; _⟨≈˘≡˘⟩_ to _⟨≈A˘≡˘⟩_
    )

module SetoidB {i j : Level} (S : Setoid i j) = Setoid' S renaming
    ( ℓ to ℓB ; Carrier to B₀ ; _≈_ to _≈B_ ; ≈-isEquivalence to ≈B-isEquivalence
    ; ≈-isPreorder to ≈B-isPreorder ; ≈-preorder to ≈B-preorder
    ; ≈-indexedSetoid to ≈B-indexedSetoid
    ; ≈-refl to ≈B-refl ; ≈-reflexive to ≈B-reflexive ; ≈-sym to ≈B-sym
    ; ≈-trans to ≈B-trans ; ≈-trans₁ to ≈B-trans₁ ; ≈-trans₂ to ≈B-trans₂
    ; _⟨≈≈⟩_ to _⟨≈B≈⟩_ ; _⟨≈≈˘⟩_ to _⟨≈B≈˘⟩_ ; _⟨≈˘≈⟩_ to _⟨≈B˘≈⟩_
    ; _⟨≈˘≈˘⟩_ to _⟨≈B˘≈˘⟩_ ; _⟨≡≈⟩_ to _⟨≡≈B⟩_ ; _⟨≡≈˘⟩_ to _⟨≡≈B˘⟩_
    ; _⟨≡˘≈⟩_ to _⟨≡˘≈B⟩_ ; _⟨≡˘≈˘⟩_ to _⟨≡˘≈B˘⟩_ ; _⟨≈≡⟩_ to _⟨≈B≡⟩_
    ; _⟨≈≡˘⟩_ to _⟨≈B≡˘⟩_ ; _⟨≈˘≡⟩_ to _⟨≈B˘≡⟩_ ; _⟨≈˘≡˘⟩_ to _⟨≈B˘≡˘⟩_
    )

module SetoidC {i j : Level} (S : Setoid i j) = Setoid' S renaming
    ( ℓ to ℓC ; Carrier to C₀ ; _≈_ to _≈C_ ; ≈-isEquivalence to ≈C-isEquivalence
    ; ≈-isPreorder to ≈C-isPreorder ; ≈-preorder to ≈C-preorder
    ; ≈-indexedSetoid to ≈C-indexedSetoid
    ; ≈-refl to ≈C-refl ; ≈-reflexive to ≈C-reflexive ; ≈-sym to ≈C-sym
    ; ≈-trans to ≈C-trans ; ≈-trans₁ to ≈C-trans₁ ; ≈-trans₂ to ≈C-trans₂
    ; _⟨≈≈⟩_ to _⟨≈C≈⟩_ ; _⟨≈≈˘⟩_ to _⟨≈C≈˘⟩_ ; _⟨≈˘≈⟩_ to _⟨≈C˘≈⟩_
    ; _⟨≈˘≈˘⟩_ to _⟨≈C˘≈˘⟩_ ; _⟨≡≈⟩_ to _⟨≡≈C⟩_ ; _⟨≡≈˘⟩_ to _⟨≡≈C˘⟩_
    ; _⟨≡˘≈⟩_ to _⟨≡˘≈C⟩_ ; _⟨≡˘≈˘⟩_ to _⟨≡˘≈C˘⟩_ ; _⟨≈≡⟩_ to _⟨≈C≡⟩_
    ; _⟨≈≡˘⟩_ to _⟨≈C≡˘⟩_ ; _⟨≈˘≡⟩_ to _⟨≈C˘≡⟩_ ; _⟨≈˘≡˘⟩_ to _⟨≈C˘≡˘⟩_
    )
```

This keeps going to cover the alphabet `SetoidD`, `SetoidE`, `SetoidF`, ..., `SetoidZ` then we shift to subscripted versions `Setoid`$_0$, `Setoid`$_1$, ..., `Setoid`$_4$.

Next, RATH-Agda shifts to the need to calculate with setoids:

SeotoidCalc𝒟 Renamings —𝒟decorated Synonyms

```
module SetoidCalcA {i j : Level} (S : Setoid i j) where
  open SetoidA S public
  open SetoidCalc S public renaming
    ( _QED to _QEDA
    ; _≈⟨_⟩_ to _≈A⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈A˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈A≡⟨_⟩_
    ; _≈⟨⟩_ to _≈A⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈A≡˘⟨_⟩_
    ; ≈-begin_ to ≈A-begin_
    )
module SetoidCalcB {i j : Level} (S : Setoid i j) where
  open SetoidB S public
  open SetoidCalc S public renaming
    ( _QED to _QEDB
    ; _≈⟨_⟩_ to _≈B⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈B˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈B≡⟨_⟩_
    ; _≈⟨⟩_ to _≈B⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈B≡˘⟨_⟩_
    ; ≈-begin_ to ≈B-begin_
    )
module SetoidCalcC {i j : Level} (S : Setoid i j) where
  open SetoidC S public
  open SetoidCalc S public renaming
    ( _QED to _QEDC
    ; _≈⟨_⟩_ to _≈C⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈C˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈C≡⟨_⟩_
    ; _≈⟨⟩_ to _≈C⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈C≡˘⟨_⟩_
    ; ≈-begin_ to ≈C-begin_
    )
```

This keeps going to cover the alphabet `SetoidCalcD, SetoidCalcE, SetoidCalcF, ...,` `SetoidCalcZ` then we shift to subscripted versions $\texttt{SetoidCalc}_0$, $\texttt{SetoidCalc}_1$, ..., $\texttt{SetoidCalc}_4$. If we ever have more than 4 setoids in hand, or prefer other decorations, then we would need to produce similar helper modules.

Each `Setoid`$\mathcal{XXX}$ takes 10 lines, for a total of at-least 600 lines!

Indeed, such renamings bloat the library, but, unlike the Standard Library, they allow new records to be declared easily —"renaming hell" has been deferred from the user to the library designer. However, later on, in `Categoric.CompOp`, we see the variations `LocalEdgeSetoid`$\mathcal{D}$ and `LocalSetoidCalc`$\mathcal{D}$ where decoration $\mathcal{D}$ ranges over $_0$, $_1$, $_2$, $_3$, $_4$, `R`. The inconsistency in not providing the other decorations used for `Setoid`$\mathcal{D}$ earlier is understandable: These take time to write and maintain.

3.2.3 Renaming Problems from the Agda-categories Library

With RATH-Agda's focus on notational modules at one end of the spectrum, and the Standard Library's casual do-as-needed in the middle, it is inevitable that there are other equally popular libraries at the other end of the spectrum. The Agda-categories library seemingly ignored the need for meaningful names altogether. Below are a few notable instances.

◇ Functors have fields named $\texttt{F}_0$, $\texttt{F}_1$, `F-resp-≈`, ...

- This could be considered reasonable even if one has a functor named `G`.
- This leads to expressions such as `< F.`F_0 `, G.`F_0 `>`.
- Incidentally, and somewhat inconsistently, a `Pseudofunctor` has fields P_0, P_1, `P-homomophism` —where the latter is documented *P preserves* ≃.

On the opposite extreme, RATH-Agda's focus on naming has its functor record with fields named `obj, mor, mor-cong` instead of F_0, F_1, `F-resp-≈` —which refer to a functor's "obj"ect map, "mor"phism map, and the fact that the "mor"phism map is a "cong"ruence.

◇ Such lack of concern for naming might be acceptable for well-known concepts such as functors, where some communities use F_i to denote the object/0-cell or morphism/1-cell operations. However, considering subcategories one sees field names `U, R, Rid, _∘R_` which are wholly unhelpful. Instead, more meaningful names such as `embed, keep, id-kept, keep-resp-∘` could have been used.

◇ The `Iso, Inverse,` and `NaturalIsomorphism` records have fields `to / from, f / f`$^{-1}$, and `F ⇒ G / F ⇐ G`, respectively.

Even though some of these build on one another, with Agda's namespacing features, all "forward" and "backward" morphism fields could have been named, say, `to` and `from`. The naming may not have propagated from `Iso` to other records possibly due to the low priority for names.

From a usability perspective, projections like `f` are reminiscent of the OCaml community and may be more acceptable there. Since Agda is more likely to attract Haskell programmers than OCaml ones, such a particular projection seems completely out of place. Likewise, the field name `F ⇒ G` seems only appropriate if the functors involved happen to be named `F` and `G`.

These unexpected deviations are not too surprising since the Agda-categories library seems to give names no priority at all. Field projections are treated little more than classic array indexing with numbers.

By largely avoiding renaming, Agda-categories has no "renaming hell" anywhere at the heavy price of being difficult to read: Any attempt to read code requires one to "squint away" the numerous projections to "see" the concepts of relevance. Consider the following excerpt.

Symbol Soup

```
helper : ∀ {F : Functor (Category.op C) (Setoids ℓ e)}
                  {A B : Obj} (f : B  ⇒  A)
                  (β γ : NaturalTransformation Hom[ C ][-, A ] F) →
                Setoid._≈_ (F₀ Nat[Hom[C][-,c],F] (F , A)) β γ →
                Setoid._≈_ (F₀ F B) (η β B ⟨$⟩ f ∘ id) (F₁ F f ⟨$⟩ (η γ A ⟨$⟩ id))
        helper {F} {A} {B} f β γ β≈γ = S.begin
          η β B ⟨$⟩ f ∘ id          S.≈⟨ cong (η β B) (id-comm ∘ (⟺
↪   identityˡ)) ⟩
          η β B ⟨$⟩ id ∘ id ∘ f     S.≈⟨ commute β f CE.refl ⟩
          F₁ F f ⟨$⟩ (η β A ⟨$⟩ id) S.≈⟨ cong (F₁ F f) (β≈γ CE.refl) ⟩
          F₁ F f ⟨$⟩ (η γ A ⟨$⟩ id) S.  □
          where module S where
                  open Setoid (F₀ F B) public
                  open SetoidR (F₀ F B) public
```

Here are a few downsides of not renaming:

1. The type of the function is difficult to comprehend; though it need not be.
   - Take `_≈₀_ = Setoid._≈_ (F₀ Nat[Hom[C][-,c],F] (F , A))`, and
   - Take `_≈₁_ = Setoid._≈_ (F₀ F B)`,
   - Then the type says: If β `≈₀` γ then `η β B ⟨$⟩ f ∘ id ≈₁ F₁ F f ⟨$⟩ (η γ A ⟨$⟩ id)` —a naturality condition!
2. The short proof is difficult to read!
   - The repeated terms such as `η β B` and `η β A` could have been renamed with mnemoic-names such as η_1, η_2 or η_s, η_t for 's'ource/1 and 't'arget/2.
   - Recall that functors `F` have projections $\mathtt{F}_i$, so the "mor"phism map on a given morphism `f` becomes `F₁ F f`, as in the excerpt above; however, using RATH-Agda's naming it would have been `mor F f`.

Since names are given a lower priority, one no longer needs to perform renaming. Instead, one is content with projections. The downside is now there are too many projections, leaving code difficult to comprehend. Moreover, this leads to inconsistent renaming.

3.3 From Is$\mathcal{X}$ to $\mathcal{X}$ —Packing away components

The distributivity axiom from earlier required an unbundled structure *after* a completely bundled structure was initially presented. Usually structures are rather large and have libraries built around them, so building and using an alternate form is not practical. However, multiple forms are usually desirable.

To accommodate the need for both forms of structure, Agda's Standard Library begins with a type-level predicate such as `IsSemigroup` below, then packs that up into a record. Here is an instance, along with comments from the library.

From Is$\mathcal{X}$ to $\mathcal{X}$ —where $\mathcal{X}$ is Semigroup

```
-- Some algebraic structures (not packed up with sets, operations, etc.)
record IsSemigroup {a ℓ} {A : Set a} (≈ : Rel A ℓ)
                   (· : Op₂ A) : Set (a ⊔ ℓ) where
  open FunctionProperties ≈
  field
    isEquivalence : IsEquivalence ≈
    assoc         : Associative ·
    ·-cong        : · Preserves₂ ≈ ⟶ ≈ ⟶ ≈

-- Definitions of algebraic structures like monoids and rings (packed in records
-- together with sets, operations, etc.)
record Semigroup c ℓ : Set (suc (c ⊔ ℓ)) where
  infixl 7 _·_
  infix  4 _≈_
  field
    Carrier     : Set c
    _≈_         : Rel Carrier ℓ
    _·_         : Op₂ Carrier
    isSemigroup : IsSemigroup _≈_ _·_
```

Listing 1: From the Agda Standard Library on Algebra

If we refer to the former as `Is`$\mathcal{X}$ and the latter as $\mathcal{X}$, then we can see similar instances in the standard library for $\mathcal{X}$ being: `Monoid`, `Group`, `AbelianGroup`, `CommutativeMonoid`, `SemigroupWithoutOne`, `NearSemiring`, `Semiring`, `CommutativeSemiringWithoutOne`, `CommutativeSemiring`, `CommutativeRing`.

It thus seems that to present an idea $\mathcal{X}$, we require the same amount of space to present it unpacked or packed, and so doing both **duplicates the process** and only hints at the underlying principle: From `Is`$\mathcal{X}$ we pack away the carriers and function symbols to obtain $\mathcal{X}$. The converse approach, starting from $\mathcal{X}$ and going to `Is`$\mathcal{X}$ is not practical, as it leads to numerous unhelpful reflexivity proofs.

Predicate Design Pattern

Present a concept $\mathcal{X}$ first as a predicate `Is`$\mathcal{X}$ on types and function symbols, then as a type $\mathcal{X}$ consisting of types, function symbols, and a proof that together they satisfy the `Is`$\mathcal{X}$ predicate.

Σ **Padding Anti-Pattern**: Starting from a bundled up type $\mathcal{X}$ consisting of types, function symbols, and how they interact, one may form the type Σ `X :` $\mathcal{X}$ `•` $\mathcal{X}$`.f X ≡ f`$_0$ to specialise the feature $\mathcal{X}$`.f` to the particular choice `f`$_0$. However, nearly all uses of this type will be of the form `(X , refl)` where the proof is unhelpful noise.

Since the standard library uses the predicate pattern, `Is`$\mathcal{X}$, which requires all sets and function symbols, the Σ-padding anti-pattern becomes a necessary evil. Instead, it would be preferable to have the family $\mathcal{X}_i$ which is the same as `Is`$\mathcal{X}$ but only takes i-many elements — c.f., `Magma`$_0$ and `Magma`$_1$ above. However, writing these variations and the necessary functions to move between them is not only tedious but also error prone. Later on, also demonstrated in [ACK19], we shall show how the bundled form $\mathcal{X}$ acts as ***the*** definition, with other forms being derived-as-needed.

Incidentally, the particular choice $\mathcal{X}_1$, a predicate on one carrier, deserves special attention. In Haskell, instances of such a type are generally known as *typeclass instances* and $\mathcal{X}_1$ is known as a *typeclass*. As discussed earlier, in Agda, we may mark such implementations for instance search using the keyword `instance`.

Typeclass Design Pattern

Present a concept $\mathcal{X}$ as a unary predicate $\mathcal{X}_1$ that associates functions and properties with a given type. Then, mark all implementations with `instance` so that arbitrary $\mathcal{X}$-terms may be written without having to specify the particular instance.

As discussed in section ???, when there are multiple instance of an $\mathcal{X}$-structure on a particular type, only one of them may be marked for instance search in a given scope.

3.4 Redundancy, Derived Features, and Feature Exclusion

A tenet of software development is not to over-engineer solutions. For example, if we need a notion of untyped composition, we may use `Monoid`. However, at a later stage, we may realise that units are inappropriate and so we need to drop them to obtain the weaker notion of `Semigroup` —for instance, if we wish to model finite functions as hashmaps, we need to omit the identity functions since they may have infinite domains; and we cannot simply enforce a convention, say, to treat empty hashmaps as the identities since then we would lose the empty functions. In weaker languages, we could continue to use the monoid interface at the cost of "throwing an exception" whenever the identity is used. However, this breaks the *Interface Segregation Principle: Users should not be forced to bother with features they are not interested in* [Mar92]. A prototypical scenario is exposing an expressive interface, possibly with redundancies, to users, but providing a minimal self-contained counterpart by dropping some features for the sake of efficiency or to act as a "smart constructor" that takes the least amount of data to reconstruct the rich interface.

More concretely, in the Agda-categories library one finds concepts with expressive interfaces, with redundant features, prototypically named $\mathcal{X}$, along with their minimal self-contained versions, prototypically named $\mathcal{X}$`Helper`. In particular, the Category type and the natural isomorphism type are instances of such a pattern. The redundant features are there to make the lives of users easier; e.g., quoting Agda-categories, *We add a symmetric proof of associativity so that the opposite category of the opposite category is definitionally equal to the original category.* To underscore the intent, we present below a minimal setup needed to express the issue. The semigroup definition contains a redundant associativity axiom —which can be obtained from the first one by applying symmetry of equality. This is done purposefully so that the "opposite, or dual, transformer" `_˘` is self-inverse on-the-nose; i.e., definitionally rather than propositionally equal. Definitionally equality does not need to be 'invoked', it is used silently when needed, thereby making the redundant setup worth it.

Redundancy can lead to silently used equalities

```
record Semigroup : Set₁ where
  constructor 𝒮
  field
    Carrier : Set
    _⨾_     : Carrier → Carrier → Carrier
    assocʳ : ∀ {x y z} →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)
    assocˡ : ∀ {x y z} →  x ⨾ (y ⨾ z)  ≡  (x ⨾ y) ⨾ z

    -- Notice:  assocˡ ≈ sym assocʳ

_˘ : Semigroup → Semigroup
(𝒮 Carrier _⨾_ assocʳ assocˡ) ˘  =  𝒮 Carrier (λ b a → a ⨾ b)  assocˡ assocʳ

˘˘≈id : ∀ {S} → (S ˘) ˘ ≡ S
˘˘≈id = refl
```

On-the-nose Redundancy Design Pattern (Agda-Categories)

Include redundant features if they allow certain common constructions to be definitionally equal, thereby requiring no overhead to use such an equality. Then, provide a smart constructor so users are not forced to produce the redundant features manually.

Incidentally, since this is not a library method, inconsistencies are bound to arise; in particular, in the $\mathcal{X}$ and $\mathcal{X}$`Helper` naming scheme: The `NaturalIsomorphism` type has `NIHelper` as its minimised version, and the type of symmetric monoidal categories is oddly called `Symmetric'` with its helper named `Symmetric`. Such issues could be reduced, if not avoided, if library methods could have been used instead.

It is interesting to note that duality forming operators, such as `_˘` above, are a design pattern themselves. How? In the setting of algebraic structures, one picks an operation to have its arguments flipped, then systematically 'flips' all proof obligations via a user-provided symmetry operator. We shall return to this as a library method in a future section.

Another example of purposefully keeping redundant features is for the sake of efficiency.

> For division semi-allegories, even though right residuals, restricted residuals, and symmetric quotients all can be derived from left residuals, we still assume them all as primitive here, since this produces more readable goals, and also makes connecting to optimised implementations easier. —RATH-Agda section 15.13

For instance, the above semigroup type could have been augmented with an ordering if we view `_⨾_` as a meet-operation. Instead, we lift such a derived operation as a primitive field, in case the user has a better implementation.

Simulating Default Implementations with Smart Constructors

```
record Order (S : Semigroup) : Set₁ where
  open Semigroup S public
  field
    _⊑_    : Carrier → Carrier → Set
    ⊑-def  : ∀ {x y} → (x ⊑ y) ≡ (x ⨾ y ≡ x)

  {- Results about _⨾_ and _⊑_ here ... -}

defaultOrder : ∀ S → Order S
defaultOrder S = let open Semigroup S
                 in record { _⊑_ = λ x y → x ⨾ y ≡ x ; ⊑-def = refl }
```

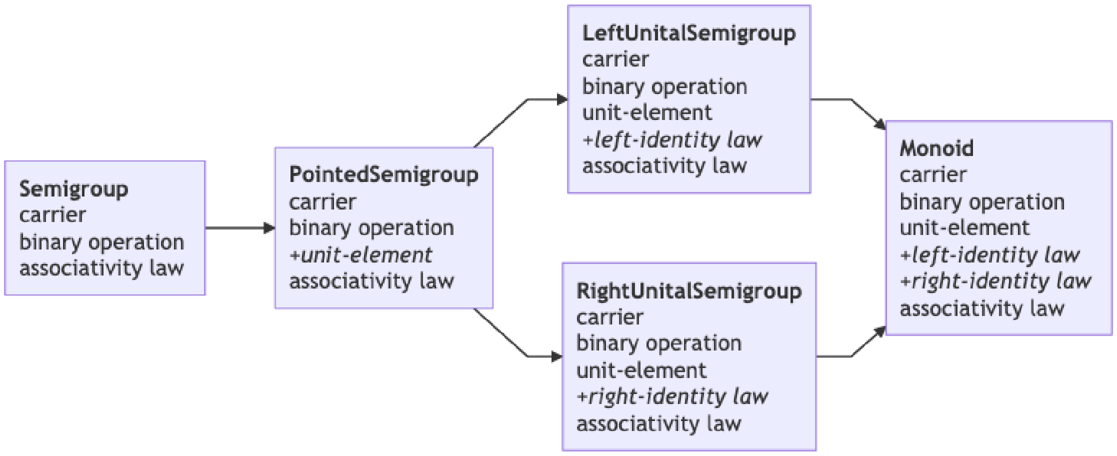
Efficient Redundancy Design Pattern (RATH-Agda section 17.1)

To enable efficient implementations, replace derived operators with additional fields for them and for the equalities that would otherwise be used as their definitions. Then, provide instances of these fields as derived operators, so that in the absence of more efficient implementations, these default implementations can be used with negligible penalty over a development that defines these operators as derived in the first place.

3.5 Extensions

In our previous discussion, we needed to drop features from `Monoid` to get `Semigroup`. However, excluding the unit-element from the monoid also required excluding the identity laws. More generally, all features reachable, via occurrence relationships, must be dropped when a particular feature is dropped. In some sense, a generated graph of features needs to be "ripped out" from the starting type, and the generated graph may be the whole type. As such, in general, we do not know if the resulting type even has any features.

Instead, in an ideal world, it is preferable to begin with a minimal interface then *extend* it with features as necessary. E.g., begin with `Semigroup` then add orthogonal features until `Monoid` is reached. Extensions are also known as *subclassing* or *inheritance*.



The libraries mentioned thus far generally implement extensions in this way. By way of example, here is how monoids could be built directly from semigroups along a particular path in the above hierarchy.

Extending Semigroup to Obtain Monoid

```
record Semigroup : Set₁ where
  field
    Carrier : Set
    _⨾_     : Carrier → Carrier → Carrier
    assoc  : ∀ {x y z} →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)

record PointedSemigroup : Set₁ where
  field semigroup : Semigroup
  open  Semigroup semigroup public {- (⋆) -}
  field Id : Carrier

record LeftUnitalSemigroup : Set₁ where
  field pointedSemigroup : PointedSemigroup
  open  PointedSemigroup pointedSemigroup public {- (⋆) -}
  field leftId : ∀ {x} → Id ⨾ x ≡ x

record Monoid : Set₁ where
  field leftUnitalSemigroup : LeftUnitalSemigroup
  open LeftUnitalSemigroup leftUnitalSemigroup public {- (⋆) -}
  field rightId : ∀ {x} → x ⨾ Id ≡ x

open Monoid  {- (⋆, *) -}

neato : ∀ {M} → Carrier M → Carrier M → Carrier M
neato {M} = _⨾_ M    {- (*); Possible due to all of the (⋆) above -}
```

Extension Design Pattern

To extend a structure $\mathcal{X}$ by new features `f`$_0$, …, `f`$_n$ which may mention features of $\mathcal{X}$, make a new structure $\mathcal{Y}$ with fields for $\mathcal{X}$, `f`$_0$, …, `f`$_n$. Then publicly open $\mathcal{X}$ in this new structure —see (⋆) above— so that the features of $\mathcal{X}$ are visible directly from $\mathcal{Y}$ to all users —see lines marked (*) above.

Notice how we accessed the binary operation `_⨾_` feature from `Semigroup` as if it were a native feature of `Monoid`. Unfortunately, `_⨾_` is only **superficially native** to `Monoid` —any actual instance, such as `woah` below, needs to define the binary operation in a `Semigroup` instance first, which lives in a `PointedSemigroup` instance, which lives in a `LeftUnitalSemigroup` instance.

Extensions are not flattened inheritance

```
woah : Monoid
woah = record { leftUnitalSemigroup
                 = record { pointedSemigroup
                             = record { semigroup = record { Carrier = {!!}
                                                           ; _⨾_     = {!!}
                                                           ; assoc   = {!!}
                                                           } -- Nesting level 3
                                      ; Id = {!!}
                                      } -- Nesting level 2
                          ; leftId = {!!}
                          } -- Nesting level 1
              ; rightId = {!!}
              }  -- Nesting level 0
```

This nesting scenario happens rather often, in one guise or another. The amount of syntactic noise required to produce a simple instantiation is unreasonable: **One should not be forced to work through the hierarchy if it provides no immediate benefit.**

Even worse, pragmatically speaking, to access a field deep down in a nested structure results in overtly lengthy and verbose names; as shown below. Indeed, in the above example, the monoid operation lives at the top-most level, we would need to access all the intermediary levels to simply refer to it. Such verbose invocations would immediately give way to helper functions to refer to fields lower in the hierarchy; yet another opportunity for boilerplate to leak in.

Extensions are not flattened inheritance

```
{- Without the (⋆) “public” declarations, projections are difficult! -}
carrier : Monoid → Set
carrier M = Semigroup.Carrier
              (PointedSemigroup.semigroup
                (LeftUnitalSemigroup.pointedSemigroup
                  (Monoid.leftUnitalSemigroup M)))
```

While library designers may be content to build `Monoid` out of `Semigroup`, users should not be forced to learn about how the hierarchy was built. Even worse, when the library designers decide to incorporate, say, `RightUnitalSemigroup` instead of the left unital form, then all users' code would break. Instead, it would be preferable to have a 'flattened' presentation for the users that “does not leak out implementation details”. We shall return to this in a future section.

It is interesting to note that diamond hierarchies cannot be trivially eliminated when providing fine-grained hierarchies. As such, we make no rash decisions regarding limiting them —and completely forgoe the unreasonable possibility of forbidding them.

A more common example from programming is that of providing monad instances in

Haskell. Most often users want to avoid tedious case analysis or prefer a sequential-style approach to producing programs, so they want to furnish a type constructor with a monad instance in order to utilise Haskell's `do`-notation. Unfortunately, this requires an applicative instances, which in turn requires a functor instance. However, providing the return-and-bind interface for monads allows us to obtain functor and applicative instances. Consequently, many users simply provide local names for the return-and-bind interface then use that to provide the default implementations for the other interfaces. In this scenario, **the standard approach is side-stepped** by manually carrying out a mechanical and tedious set of steps that not only wastes time but obscures the generic process and could be error-prone.

Instead, it would be desirable to 'flatten' the hierarchy into a single package, consisting of the fields throughout the hierarchy, possibly with default implementations, yet still be able to view the resulting package at base levels in the hierarchy —c.f., section 3.4. Another benefit of this approach is that it allows users to utilise the package without consideration of how the hierarchy was formed, thereby providing library designers with the freedom to alter it in the future.

3.6 Conclusion

After 'library spelunking', we are now in a position to summarise the problems encountered, when using existing[3] modules systems, that need a solution. From our learned lessons, we can then pinpoint a necessary feature of an ideal module system for dependently-typed languages.

3.6.1 Lessons Learned

Systems tend to come with a pre-defined set of operations for built-in constructs; the user is left to utilise third-party pre-processing tools, for example, to provide extra-linguistic support for common repetitive scenarios they encounter.

More concretely, a large number of proofs can be discharged by merely pattern matching on variables —this works since the case analysis reduces the proof goal into a trivial reflexitivity obligation, for example. The number of cases can quickly grow thereby taking up space, which is unfortunate since the proof has very little to offer besides verifying the claim. In such cases, a pre-process, perhaps an "editor tactic", could be utilised to produce the proof in an auxiliary file, and reference it in the current file.

Perhaps more common is the renaming of package contents, by hand. For example, when a notion of preorder is defined with relation named `_≤_`, one may rename it and all references to it by, say, `_⊑_`. Again, a pre-processor or editor-tactic could be utilised, but many simply perform the re-write by hand —which is tedious, error prone, and obscures the

[3]A comparison of module systems of other dependently-typed languages is covered in section **??**.

generic rewriting method.

It would be desirable to **allow packages to be treated as first-class concepts that could be acted upon, in order to avoid third-party tools that obscure generic operations and leave them out of reach for the powerful typechecker of a dependently typed system.** Below is a summary of the design patterns mentioned above, using monoids as the prototypical structure. Some patterns we did not cover, as they will be covered in future sections.

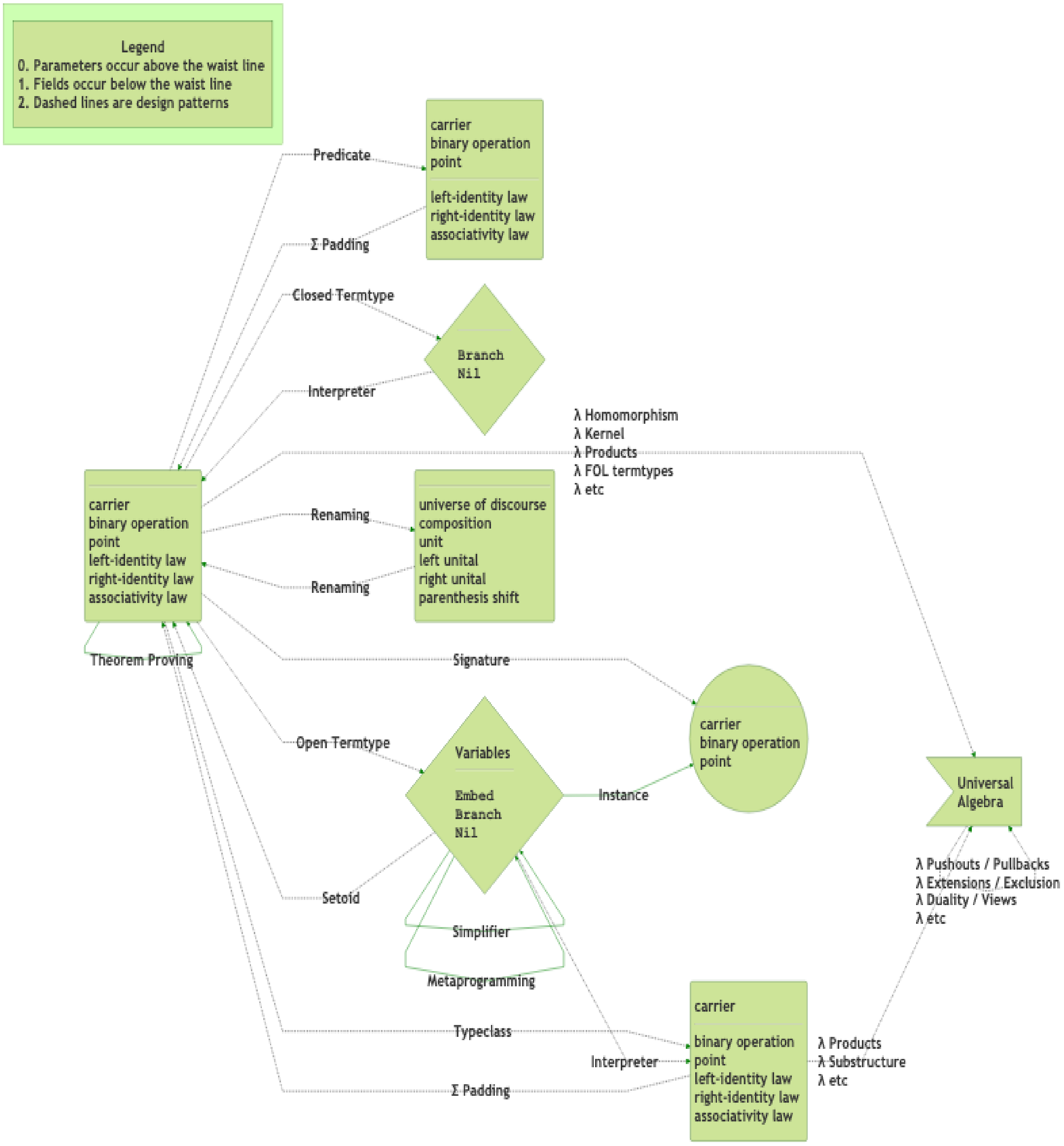


Figure 3.1: PL Research is about getting free stuff: From the left-most node, we can get a lot!

Remarks:

1. It is important to note that the `termtype` constructions could also be co-inductive, thereby yielding possibly infinitely branching syntax-trees.

   ◇ In the "simplify" pattern, one could use axioms as rewrite rules.

2. It is more convenient to restrict a carrier or to form products along carriers using the typeclass version.

3. As discussed earlier, the name *typeclass* is justified not only by the fact that this is the shape used by typeclasses in Haskell and Coq, but also that instance search for such records is supported in Agda by using the `instance` keyword.

There are many more design patterns in dependently-typed programming. Since grouping mechanisms are our topic, we have only presented those involving organising data.

3.6.2 One-Item Checklist for a Candidate Solution

An adequate module system for dependently-typed languages should make use of dependent-types as much as possible. As such, there is essentially one and only one primary goal for a module system to be considered reasonable for dependently-typed languages: Needless distinctions should be eliminated as much as possible.

The "write once, instantiate many" attitude is well-promoted in functional communities predominately for *functions*, but we will take this approach to modules as well, beyond the features of, e.g., SML functors. With one package declaration, one should be able to mechanically derive data, record, typeclass, product, sum formulations, among many others. All operations on the generic package then should also apply to the particular package instantiations.

This one goal for a reasonable solution has a number of important and difficult subgoals. The resulting system should be well-defined with a coherent semantic underpinning —possibly being a conservative extension—; it should support the elementary uses of pedestrian module systems; the algorithms utilised need to be proven correct with a mechanical proof assistant, considerations for efficiency cannot be dismissed if the system is to be usable; the interface for modules should be as minimal as possible, and, finally, a large number of existing use-cases must be rendered tersely using the resulting system without jeopardising runtime performance in order to demonstrate its success.

Bibliography

[20] *Agda Standard Library*. 2020. URL: `https://github.com/agda/agda-stdlib` (visited on 03/03/2020) (cit. on p. 8).

[ACK19] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. “A language feature to unbundle data at will (short paper)”. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*. Ed. by Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm. ACM, 2019, pp. 14–19. ISBN: 978-1-4503-6980-0. DOI: `10.1145/3357765.3359523`. URL: `https://doi.org/10.1145/3357765.3359523` (cit. on p. 17).

[Jac20] Jason Hu Jacque Carrette. *agda-categories library*. 2020. URL: `https://github.com/agda/agda-categories` (visited on 08/20/2020) (cit. on p. 8).

[Kah18] Wolfram Kahl. *Relation-Algebraic Theories in Agda*. 2018. URL: `http://relmics.mcmaster.ca/RATH-Agda/` (visited on 10/12/2018) (cit. on p. 8).

[Mar92] Robert C. Martin. *Design Principles and Design Patterns*. Ed. by Deepak Kapur. 1992. URL: `https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf` (visited on 10/19/2018) (cit. on p. 18).