The Final Supervisory Committee Meeting

Musa Al-hassy

May 20, 2020

Past and Present Efforts

Research Question

Use a dependently-typed language (DTL) to implement the ‘missing’ module system features directly inside the language

Research Question

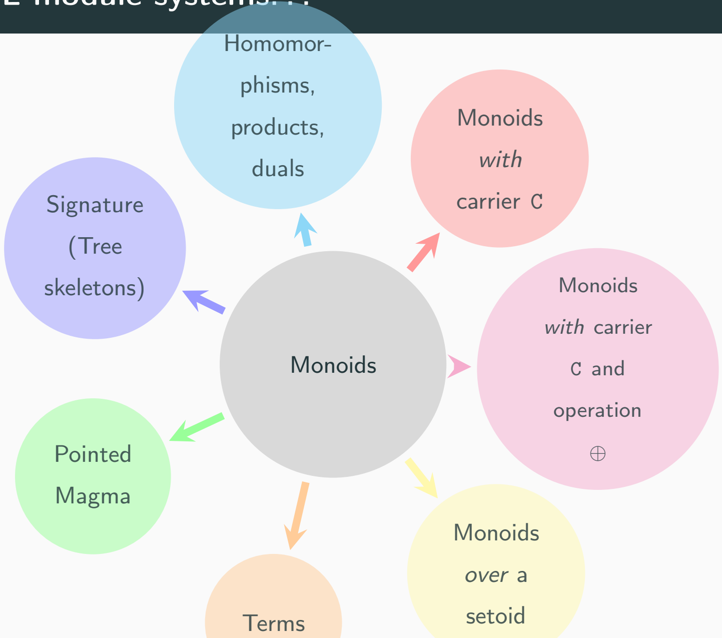
Use a dependently-typed language (DTL) to implement the ‘missing’ module system features directly inside the language

---

```
-- Terms: Expressions and Types
e, τ ::= α            -- base types and constants
    | Typeᵢ        -- “type of types”; Universe of types at level i : ℕ
    | ℕ             -- “Levels” for the type hierarchy
    | Π x : τ • τ    -- “Pi”, dependent-function type
    | Σ x : τ • τ    -- “Sigma”, dependent-sum type
    | x             -- Variable
    | e e           -- Application; Π-elimination
    | λ x : τ • e    -- Abstraction; Π-introduction
    | (e , e)       -- Pairing; Σ-introduction
    | fst e | snd e -- Projections; Σ-elimination
    | Fix F         -- Fixpoints for F : Typeᵢ → Typeᵢ

-- Abbreviation: Provided β does not refer to variable ‘_’,
(α → β) :=  (Π _ : α • β)
```

Ubiquitous mechanical module constructions are out of reach of DTL module systems. . .



Evidence that the theory 'actually works'

Prototype with an editor extension *then* incorporate lessons learned into a DTL library!

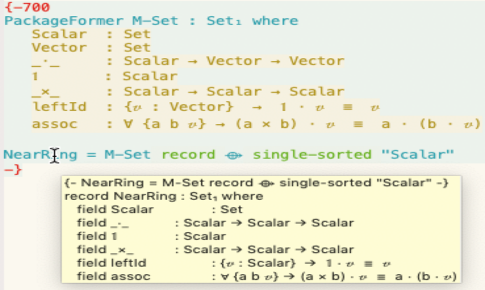
```
{-700
PackageFormer M-Set : Set₁ where
    Scalar  : Set
    Vector  : Set
    _·_     : Scalar → Vector → Vector
    1       : Scalar
    _×_     : Scalar → Scalar → Scalar
    leftId  : {𝓋 : Vector} → 1 · 𝓋 ≡ 𝓋
    assoc   : ∀ {a b 𝓋} → (a × b) · 𝓋 ≡ a · (b · 𝓋)

NearRing = M-Set record ⟴ single-sorted "Scalar"
-}
```

```
{- NearRing = M-Set record ⟴ single-sorted "Scalar" -}
record NearRing : Set₁ where
  field Scalar        : Set
  field _·_       : Scalar → Scalar → Scalar
  field 1         : Scalar
  field _×_       : Scalar → Scalar → Scalar
  field leftId        : {𝓋 : Scalar} → 1 · 𝓋 ≡ 𝓋
  field assoc         : ∀ {a b 𝓋} → (a × b) · 𝓋 ≡ a · (b · 𝓋)
```

Evidence that the theory 'actually works'

Prototype with an editor extension *then* incorporate lessons learned into a DTL library!



1. `PackageFormer` Emacs Editor Extension
2. `Context` Agda Library

Prototype ⇒ Publication

A Language Feature to Unbundle Data at Will (GPCE '19)

grammer sufficient tools to adequately express such ideas. As such, for the rest of this paper we will illustrate our ideas in Agda [2, 7]. For the monoid example, it seems that there are three contenders for the monoid interface:

```
record Monoid₀ : Set₁ where
  field
    Carrier : Set
    _⨾_     : Carrier → Carrier → Carrier
    Id      : Carrier
    assoc   : ∀ {x y z}
            → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x

record Monoid₁ (Carrier : Set) : Set where
  field
    _⨾_     : Carrier → Carrier → Carrier
    Id      : Carrier
    assoc   : ∀ {x y z}
            → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x

record Monoid₂
         (Carrier : Set)
         (_⨾_ : Carrier → Carrier → Carrier)
       : Set  where
  field
    Id      : Carrier
    assoc   : ∀ {x y z}
            → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x
```

In Monoid₀, we will call Carrier "bundled up", while we call it "exposed" in Monoid₁ and Monoid₂. The bundled-up version allows us to speak of *a* monoid, rather than *a monoid on a given type* which is captured by Monoid₁. While Monoid₂ exposes both the carrier and the composition operation, we

syntax. For example, the syntax of closed monoid terms can be expressed, using trees, as follows.

```
data Monoid₃ : Set where
  _⨾_ : Monoid₃ → Monoid₃ → Monoid₃
  Id  : Monoid₃
```

We can see that this can be obtained from Monoid₀ by discarding the fields denoting equations, then turning the remaining fields into constructors.

We show how these different presentations can be derived from a *single* PackageFormer declaration via a generative meta-program integrated into the most widely-used Agda "IDE", the Emacs mode for Agda. In particular, if one were to explicitly write M different bundlings of a package with N constants then one would write nearly $N \times M$ lines of code, yet this quadratic count becomes linear $N + M$ by having a single package declaration of N constituents with M subsequent instantiations. We hope that reducing such duplication of effort, and of potential maintenance burden, will be beneficial to the software engineering of large libraries of formal code — and consider it the main contribution of our work.

2 PackageFormers — Being Non-committal as Much as Possible

We claim that the above monoid-related pieces of Agda code can be unified as a single declaration which does not distinguish between parameters and fields, where PackageFormer is a keyword with similar syntax as record:

```
PackageFormer MonoidP : Set₁ where
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
  leftId  : ∀ {x} → Id ⨾ x ≡ x
  rightId : ∀ {x} → x ⨾ Id ≡ x
```

(For clarity, this and other non-native Agda syntax is left uncoloured.)

Prototype ⇒ Lisp Metaprogramming, ASTs, Untyped, String Manipulation, Agda Generation, Macro DSL

```
{-lisp
(𝒱 record₁ (discard-equations nil)
 = "Reify a variational as an Agda ‘‘record’’.
    Elements with equations are construed as
    derivatives of fields  ---the elements
    without any equations--- by default, unless
    DISCARD-EQUATIONS is provided with a non-nil value."
  :kind record
  :alter-elements
    (λ es →
      (thread-last es
      ;; Keep or drop eqns depending on “discard-equations”
      (--map
        (if discard-equations
            (map-equations (λ _ → nil) it)
            it))
      ;; Unless there’s equations, mark elements as fields.
      (--map (map-qualifier
        (λ _ → (unless (element-equations it)
               "field")) it)))))
-}

{-700
Monoid-record-with-definitional-extensions  =  MonoidP record₁
Monoid-record-with-extensions-as-fields     =  MonoidP record₁ :discard-equations t
-}
```

Generated 200+ theories using the Lisp metaprogramming framework —the MathScheme library

```
AdditiveMagma             = Magma renaming’ "_*_ to _+_"
LeftDivisionMagma         = Magma renaming’ "_*_ to _\_"
RightDivisionMagma        = Magma renaming’ "_*_ to _/_"
LeftOperation             = MultiCarrier extended-by’ "_⟫_ : U → S → S"
RightOperation            = MultiCarrier extended-by’ "_⟪_ : S → U → S"
IdempotentMagma           = Magma extended-by’ "*-idempotent : ∀ (x : U) → (x * x) ≡ x"
IdempotentAdditiveMagma   = IdempotentMagma renaming’ "_*_ to _+_"
SelectiveMagma            = Magma extended-by’ "*-selective : ∀ (x y : U) → (x * y ≡ x) ⊎ (x * y ≡ y)"
SelectiveAdditiveMagma    = SelectiveMagma renaming’ "_*_ to _+_"
PointedMagma              = Magma union’ PointedCarrier
Pointed𝟘Magma              = PointedMagma renaming’ "e to 𝟘"
AdditivePointed1Magma     = PointedMagma renaming’ "_*_ to _+_; e to 𝟙"
LeftPointAction           = PointedMagma extended-by "pointactLeft  :  U → U; pointactLeft x = e * x"
RightPointAction          = PointedMagma extended-by "pointactRight  :  U → U; pointactRight x = x * e"
CommutativeMagma          = Magma extended-by’ "*-commutative  :  ∀ (x y : U) →  (x * y) ≡ (y * x)"
CommutativeAdditiveMagma  = CommutativeMagma renaming’ "_*_ to _+_"
PointedCommutativeMagma   = PointedMagma union’ CommutativeMagma ⟴  :remark "over Magma"
AntiAbsorbent             = Magma extended-by’ "*-anti-self-absorbent  : ∀ (x y : U) → (x * (x * y)) ≡ y"
SteinerMagma              = CommutativeMagma union’ AntiAbsorbent ⟴  :remark "over Magma"
Squag                     = SteinerMagma union’ IdempotentMagma ⟴  :remark "over Magma"
PointedSteinerMagma       = PointedMagma union’ SteinerMagma ⟴  :remark "over Magma"
UnipotentPointedMagma     = PointedMagma extended-by’ "unipotent  : ∀ (x : U) →  (x * x) ≡ e"
Sloop                     = PointedSteinerMagma union’ UnipotentPointedMagma
```

Generated 200+ theories using the Lisp metaprogramming framework —the MathScheme library

```
AdditiveMagma              = Magma renaming’ "_*_ to _+_"
LeftDivisionMagma          = Magma renaming’ "_*_ to _\_"
RightDivisionMagma         = Magma renaming’ "_*_ to _/_"
LeftOperation
RightOperation
IdempotentMagma
IdempotentAdditive
SelectiveMagma             = Magma extended-by’ "*-selective : ∀ (x y : U) → (x * y ≡ x) ⊎ (x * y ≡ y)"
SelectiveAdditiveMagma     = SelectiveMagma renaming’ "_*_ to _+_"
PointedMagma               = Magma union’ PointedCarrier
Pointed𝟘Magma               = PointedMagma renaming’ "e to 𝟘"
AdditivePointed1Magma      = PointedMagma renaming’ "_*_ to _+_; e to 𝟙"
LeftPointAction            = PointedMagma extended-by "pointactLeft  :  U → U; pointactLeft x = e * x"
RightPointAction           = PointedMagma extended-by "pointactRight  :  U → U; pointactRight x = x * e"
CommutativeMagma           = Magma extended-by’ "*-commutative  :  ∀ (x y : U) →  (x * y) ≡ (y * x)"
CommutativeAdditiveMagma = CommutativeMagma renaming’ "_*_ to _+_"
PointedCommutativeMagma    = PointedMagma union’ CommutativeMagma —⊕  :remark "over Magma"
AntiAbsorbent              = Magma extended-by’ "*-anti-self-absorbent  : ∀ (x y : U) → (x * (x * y)) ≡ y"
SteinerMagma               = CommutativeMagma union’ AntiAbsorbent —⊕  :remark "over Magma"
Squag                      = SteinerMagma union’ IdempotentMagma —⊕  :remark "over Magma"
PointedSteinerMagma        = PointedMagma union’ SteinerMagma —⊕  :remark "over Magma"
UnipotentPointedMagma      = PointedMagma extended-by’ "unipotent  : ∀ (x : U) →  (x * x) ≡ e"
Sloop                      = PointedSteinerMagma union’ UnipotentPointedMagma
```

Terse, readable, specifications

↦ Useful, typecheckable, dauntingly large code

Generated 200+ theories using the Lisp metaprogramming framework —the MathScheme library

```
AdditiveMagma                = Magma renaming' "_*_ to _+_"
LeftDivisionMagma            = Magma renaming' "_*_ to _\_"
RightDivisionMagma           = Magma renaming' "_*_ to _/_"
LeftOperation
RightOperation
IdempotentMagma
IdempotentAdditive
SelectiveMagma               = Magma extended-by' "*-selective : ∀ (x y : U) → (x * y ≡ x) ⊎ (x * y ≡ y)"
SelectiveAdditiveMagma       = SelectiveMagma renaming' "_*_ to _+_"
PointedMagma                 = Magma union' PointedCarrier
Pointed𝟘Magma
AdditivePointed1Ma                                  +_; e to 𝟙"
LeftPointAction                                                                    x = e * x"
RightPointAction                                                                   t x = x * e"
CommutativeMagma             = Magma extended-by' "*-commutative  :  ∀ (x y : U) →  (x * y) ≡ (y * x)"
CommutativeAdditiveMagma = CommutativeMagma renaming' "_*_ to _+_"
PointedCommutativeMagma      = PointedMagma union' CommutativeMagma —⊕ :remark "over Magma"
AntiAbsorbent                = Magma extended-by' "*-anti-self-absorbent  : ∀ (x y : U) → (x * (x * y)) ≡ y"
SteinerMagma                 = CommutativeMagma union' AntiAbsorbent —⊕ :remark "over Magma"
Squag                        = SteinerMagma union' IdempotentMagma —⊕ :remark "over Magma"
PointedSteinerMagma          = PointedMagma union' SteinerMagma —⊕ :remark "over Magma"
UnipotentPointedMagma        = PointedMagma extended-by' "unipotent  : ∀ (x : U) →  (x * x) ≡ e"
Sloop                        = PointedSteinerMagma union' UnipotentPointedMagma
```

Terse, readable, specifications

↦ Useful, typecheckable, dauntingly large code

200+ **one-line** specs

↦ 1500+ lines of typechecked Agda

Generated 200+ theories using the Lisp metaprogramming framework —the MathScheme library

```
AdditiveMagma             = Magma renaming' "_*_ to _+_"
LeftDivisionMagma         = Magma renaming' "_*_ to _\_"
RightDivisionMagma        = Magma renaming' "_*_ to _/_"
LeftOperation
RightOperation
IdempotentMagma
IdempotentAdditive
SelectiveMagma            = Magma extended-by' "*-selective : ∀ (x y : U) → (x * y ≡ x) ⊎ (x * y ≡ y)"
SelectiveAdditiveMagma    = SelectiveMagma renaming' "_*_ to _+_"
PointedMagma              = Magma union' PointedCarrier
Pointed𝟘Magma
AdditivePointed1Ma                                  +_; e to 𝟙"
LeftPointAction                                                                      x = e * x"
RightPointAction                                                                    t x = x * e"
CommutativeMagma                                                 y : U) → (x * y) ≡ (y * x)"
CommutativeAdditiv
PointedCommutativeMagma   = PointedMagma union' CommutativeMagma —⊕ :remark "over Magma"
AntiAbsorbent             = Magma extended-by' "*-anti-self-absorbent : ∀ (x y : U) → (x * (x * y)) ≡ y"
SteinerMagma                                                       —⊕ :remark "over Magma"
Squag                                                            ⊕ :remark "over Magma"
PointedSteinerMagma       = PointedMagma union' SteinerMagma —⊕ :remark "over Magma"
UnipotentPointedMagma     = PointedMagma extended-by' "unipotent : ∀ (x : U) → (x * x) ≡ e"
Sloop                     = PointedSteinerMagma union' UnipotentPointedMagma
```

Terse, readable, specifications

↦ Useful, typecheckable, dauntingly large code

200+ **one-line** specs

↦ 1500+ lines of typechecked Agda

⇒ 750% efficiency savings

Useful engineering result

Pushout unions, intersections, extensions, views, ...

```
(𝒱 union pf (renaming₁ "") (renaming₂ "") (adjoin-retract₁ t) (adjoin-retract₂ t)
 = "Union the elements of the parent PackageFormer with those of
    the provided PF symbolic name, then adorn the result with two views:
    One to the parent and one to the provided PF.

    If an identifer is shared but has different types, then crash."
   :alter-elements (λ es →
     (let* ((p (symbol-name 'pf))
            (es₁ (alter-elements es renaming renaming₁ :adjoin-retract nil))
            (es₂ (alter-elements ($elements-of p) renaming renaming₂ :adjoin-retract nil))
            (es' (-concat es₁ es₂)))

      ;; Ensure no name clashes!
      (loop for n in (find-duplicates (mapcar #'element-name es'))
            for e = (--filter (equal n (element-name it)) es')
            unless (--all-p (equal (car e) it) e)
            do (-let [debug-on-error nil]
              (error "%s = %s union %s \n\n\t\t → Error: Elements ‘%s’ conflict!\n\n\t\t\t%s"
                     $name $parent p (element-name (car e)) (s-join "\n\t\t\t" (mapcar #'show-element e)))))

   ;; return value
   (-concat
       es'
       (when adjoin-retract₁ (list (element-retract $parent es :new es₁ :name adjoin-retract₁)))
       (when adjoin-retract₂ (list (element-retract p     ($elements-of p) :new es₂ :name
 ↪     adjoin-retract₂)))))))
```

Primitives are motivated from existing, real-world, DTL libraries!

Primary Lesson Learned: `:waist`

The difference between field and parameter is an illusion —as is that of input and output when one considers relations rather than deterministic functions.

Primary Lesson Learned: `:waist`

The difference between field and parameter is an illusion —as is that of input and output when one considers relations rather than deterministic functions.

User-defined variational: *Drop definitions when lifting fields into parameters.*

```
(𝒱 unbundling n
 = "Turn the first N elements into parameters to the PackageFormer.

    Any elements above the waist line have their equations dropped."
   :waist n
   :alter-elements (λ es →
     (-let [i 0]
       (--graph-map (progn (incf i) (<= i n))
                    (map-equations (-const nil) it)
                    es))))
```

Characterising `:waist` as $\Pi{\to}\lambda$

```
Π→λ (Π a : A • τ)  =  (λ a : A • τ)
```

Characterising `:waist` as $\Pi{\to}\lambda$

$$\Pi{\to}\lambda\ (\Pi\ a : A \bullet \tau) \;=\; (\lambda\ a : A \bullet \tau)$$

$id_0 : Set_1$
$id_0 = \Pi\ X : Set \bullet \Pi\ e : X \bullet X$

$id_1 : \Pi\ X : Set \bullet Set$
$id_1 = \lambda\ (X : Set) \to \Pi\ e : X \bullet X$

$id_2 : \Pi\ X : Set \bullet \Pi\ e : X \bullet Set$
$id_2 = \lambda\ (X : Set)\ (e : X) \to X$

Characterising `:waist` as $\Pi{\to}\lambda$

```
Π→λ (Π a : A • τ)  =  (λ a : A • τ)
```

```
id₀ : Set₁
id₀ = Π X : Set • Π e : X • X

id₁ : Π X : Set • Set
id₁ = λ (X : Set) → Π e : X • X

id₂ : Π X : Set • Π e : X • Set
id₂ = λ (X : Set) (e : X) → X
```

- $\mathtt{id}_{i+1} \approx \Pi{\to}\lambda\ \mathtt{id}_i$
- $\mathtt{id}_0$ is a *type of functions*
- $\mathtt{id}_1$ is a *function on types*

Context Agda Library ⇒ Pragmatic Interface

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              _⊕_     ← (Carrier → Carrier → Carrier)
              Id      ← Carrier
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

`Context` Agda Library ⇒ Pragmatic Interface

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              _⊕_     ← (Carrier → Carrier → Carrier)
              Id      ← Carrier
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

- Ideas: *Weak* Agda Reflection, No fresh names, Monads, Termination, 'Reification' $\Pi \to \lambda$

Context Agda Library ⇒ Pragmatic Interface

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              _⊕_     ← (Carrier → Carrier → Carrier)
              Id      ← Carrier
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

- Ideas: *Weak* Agda Reflection, No fresh names, Monads, Termination, 'Reification' $\Pi \to \lambda$
- Draft paper: *Do-it-yourself Module Systems*

‘All’ module constructions are born from `Context`

‘All’ module constructions are born from `Context`

- Context: “name-type pairs”

  ```
  do S ← Set; s ← S; n ← (S → S); End
  ```

‘All’ module constructions are born from `Context`

- Context: “name-type pairs”

  `do S ← Set; s ← S; n ← (S → S); End`

- Record Type: “bundled-up data”

  $\Sigma\ \mathtt{S} : \mathtt{Set} \bullet \Sigma\ \mathtt{s} : \mathtt{S} \bullet \Sigma\ \mathtt{n} : \mathtt{S} \to \mathtt{S} \bullet \mathbb{1}$

‘All’ module constructions are born from `Context`

- Context: “name-type pairs”

  `do S ← Set; s ← S; n ← (S → S); End`

- Record Type: “bundled-up data”

  Σ `S : Set` $\bullet$ Σ `s : S` $\bullet$ Σ `n : S` $\to$ `S` $\bullet$ $\mathbb{1}$

- Function Type: “a type of functions”

  Π `S` $\bullet$ Σ `s : S` $\bullet$ Σ `n : S` $\to$ `S` $\bullet$ $\mathbb{1}$

‘All’ module constructions are born from `Context`

- Context: “name-type pairs”

  `do S ← Set; s ← S; n ← (S → S); End`

- Record Type: “bundled-up data”

  `Σ S : Set • Σ s : S • Σ n : S → S • 𝟙`

- Function Type: “a type of functions”

  `Π S • Σ s : S • Σ n : S → S • 𝟙`

- Type constructor: “a function on types”

  `λ S • Σ s : S • Σ n : S → S • 𝟙`

‘All’ module constructions are born from `Context`

- Context: “name-type pairs”

  `do S ← Set; s ← S; n ← (S → S); End`

- Record Type: “bundled-up data”

  `Σ S : Set • Σ s : S • Σ n : S → S • 𝟙`

- Function Type: “a type of functions”

  `Π S • Σ s : S • Σ n : S → S • 𝟙`

- Type constructor: “a function on types”

  `λ S • Σ s : S • Σ n : S → S • 𝟙`

- Algebraic datatype: “a descriptive syntax”

  `data 𝔻 : Set where s : 𝔻; n : 𝔻 → 𝔻`

Comparing `PackageFormer` and `Context`

	PackageFormer	Contexts
Type of Entity	Preprocessing Tool	Language Library
Specification Language	Lisp + Agda	Agda
Well-formedness Checking	×	✓
Termination Checking	✓	✓
Elaboration Tooltips	✓	×
Rapid Prototyping	✓	✓ (Slower)
Usability Barrier	None	None
Extensibility Barrier	Lisp	Weak Metaprogramming

Current Activities

1. Complete a interpreter, via a rewrite-system, for `PackageFormer`

2. Finish writing thesis
   - Demonstrate that common module idioms are expressible in our framework
   - Demonstrate that several uncommon notions of packaging from universal algebra are also possible!

Contributions

Module Systems for DTLs

1. The ability to *implement* module systems for DTLs within DTLs

Module Systems for DTLs

1. The ability to *implement* module systems for DTLs within DTLs
2. The ability to arbitrarily extend such systems by users at a high-level

Module Systems for DTLs

1. The ability to *implement* module systems for DTLs within DTLs
2. The ability to arbitrarily extend such systems by users at a high-level
3. Demonstrate that there is an expressive yet minimal set of module meta-primitives which allow common module constructions to be defined

Module Systems for DTLs

1. The ability to *implement* module systems for DTLs within DTLs
2. The ability to arbitrarily extend such systems by users at a high-level
3. Demonstrate that there is an expressive yet minimal set of module meta-primitives which allow common module constructions to be defined
4. Demonstrate that relationships between modules can also be mechanically generated.

Termtypes as Modules

5. Bring algebraic data types under the umbrella of grouping mechanisms: An ADT is just a context whose symbols target the ADT ‘carrier’ and are not otherwise interpreted.
    - In particular, both an ADT and a record can be obtained practically from a single context declaration.

Termtypes as Modules

5. Bring algebraic data types under the umbrella of grouping mechanisms: An ADT is just a context whose symbols target the ADT ‘carrier’ and are not otherwise interpreted.
    - In particular, both an ADT and a record can be obtained practically from a single context declaration.

```
DynamicSystem : Context ℓ₁
DynamicSystem
    = do State ← Set
         start ← State
         next  ← (State → State)
         End
```

Termtypes as Modules

5. Bring algebraic data types under the umbrella of grouping mechanisms: An ADT is just a context whose symbols target the ADT 'carrier' and are not otherwise interpreted.
    - In particular, both an ADT and a record can be obtained practically from a single context declaration.

```
DynamicSystem : Context ℓ₁
DynamicSystem
    = do State ← Set
         start ← State
         next  ← (State → State)
         End
```

```
data 𝔻 : Set where
    startD : 𝔻
    nextD  : 𝔻 → 𝔻
```

Termtypes as Modules

5. Bring algebraic data types under the umbrella of grouping mechanisms: An ADT is just a context whose symbols target the ADT 'carrier' and are not otherwise interpreted.
   - In particular, both an ADT and a record can be obtained practically from a single context declaration.

```
DynamicSystem : Context ℓ₁
DynamicSystem
    = do State ← Set
         start ← State
         next  ← (State → State)
         End
```

```
data 𝔻 : Set where
    startD : 𝔻
    nextD  : 𝔻 → 𝔻
```

```
𝔻 = termtype (DynamicSystem :waist 1)

-- Pattern synonyms for more compact presentation
pattern startD   = μ (inj₁ tt)        -- : 𝔻
pattern nextD e = μ (inj₂ (inj₁ e)) -- : 𝔻 → 𝔻
trivial : 𝔻 ≅ ℕ
```

Common data-structures as free termtypes

6. Show that common data-structures are mechanically the (free) termtypes of common modules.

Common data-structures as free termtypes

6. Show that common data-structures are mechanically the (free) termtypes of common modules.

Module System	Termtype
Dynamical Structures	Naturals
Collection Structures	Lists
Pointed Structures	Maybe

Common data-structures as free termtypes

6. Show that common data-structures are mechanically the (free) termtypes of common modules.

Module System	Termtype
Dynamical Structures	Naturals
Collection Structures	Lists
Pointed Structures	Maybe

```
Collection : ∀ ℓ → Context (ℓsuc ℓ)
Collection ℓ = do Elem    ← Set ℓ
                  Carrier ← Set ℓ
                  insert  ← (Elem → Carrier → Carrier)
                  ∅       ← Carrier
                  End {ℓ}

List : Set → Set
List ElemType = termtype ((Collection ℓ0 :waist 2) ElemType)

pattern _::_ x xs = μ (inj1 (x , xs , tt))
pattern  ∅        = μ (inj2 (inj1 tt))
```

Solve the unbundling problem —all in Agda!

7. The ability to 'unbundle' module fields as if they were parameters 'on the fly'

Solve the unbundling problem —all in Agda!

7. The ability to 'unbundle' module fields as if they were parameters 'on the fly'

```
DynamicSystem : Context ℓ₁
DynamicSystem
    = do State ← Set
         start ← State
         next  ← (State → State)
         End
```

Solve the unbundling problem —all in Agda!

7. The ability to 'unbundle' module fields as if they were parameters 'on the fly'

```
DynamicSystem : Context ℓ₁
DynamicSystem
    = do State ← Set
         start ← State
         next  ← (State → State)
         End
```

```
𝒩⁰ : DynamicSystem :waist 0
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : (DynamicSystem :waist 1) ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : (DynamicSystem :waist 2) ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : (DynamicSystem :waist 3) ℕ 0
 ↪  suc
𝒩³ = ⟨⟩
```

Solve the unbundling problem —all in Agda!

7. The ability to 'unbundle' module fields as if they were parameters 'on the fly'

```
DynamicSystem : Context ℓ₁
DynamicSystem
    = do State ← Set
         start ← State
         next  ← (State → State)
         End
```

Without redefining DynamicSystem, we are able to **fix** some of its fields by making them into parameters!

```
𝒩⁰ : DynamicSystem :waist 0
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : (DynamicSystem :waist 1) ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : (DynamicSystem :waist 2) ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : (DynamicSystem :waist 3) ℕ 0
  ↪  suc
𝒩³ = ⟨⟩
```

Solve the unbundling problem —all in Agda!

7. The ability to 'unbundle' module fields as if they were parameters 'on the fly'

```
DynamicSystem : Context ℓ₁
DynamicSystem
    = do State ← Set
         start ← State
         next  ← (State → State)
         End
```

Without redefining `DynamicSystem`, we are able to **fix** some of its fields by making them into parameters!

```
𝒩⁰ : DynamicSystem :waist 0
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : (DynamicSystem :waist 1) ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : (DynamicSystem :waist 2) ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : (DynamicSystem :waist 3) ℕ 0
 ↪  suc
𝒩³ = ⟨⟩
```

The type of dynamic systems over carrier ℕ and start state 0 is `(DynamicSystem :waist 2) ℕ 0`.

Theory & Implementation

8. Demonstrate that there is a practical implementation of such a framework
   - ☒ The `Context` framework is implemented in Agda and we've seen practical examples of its use.

Theory & Implementation

8. Demonstrate that there is a practical implementation of such a framework
   - ☒ The `Context` framework is implemented in Agda and we've seen practical examples of its use.

9. Finally, the resulting framework is *mostly* type-theory agnostic: The target setting is DTLs but we only assume the barebones; if users drop parts of that theory, then *only* some parts of the framework will no longer apply.
   - ☐ Started . . .

Next Steps

SMART Goals

June 2020 Finish interpreter for `PackageFormer`

SMART Goals

June 2020 Finish interpreter for `PackageFormer`

July 2020 Finish writing thesis

- Possibly submit draft paper *Do-it-yourself Module Systems*

SMART Goals

June 2020 Finish interpreter for `PackageFormer`

July 2020 Finish writing thesis

- Possibly submit draft paper *Do-it-yourself Module Systems*

August 2020 Defend thesis

Summary

Summary

1. Published one paper regarding research and have a draft ready to be cleaned-up
2. Currently working on the thesis with the intention of defending in the next few months

Summary

1. Published one paper regarding research and have a draft ready to be cleaned-up

2. Currently working on the thesis with the intention of defending in the next few months

Thank-you for your time!

Questions?