Do-it-yourself Module Systems

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

PhD Defence

Musa Al-hassy

April 28, 2021

McMaster University, Hamilton, Ontario, Canada
alhassy@gmail.com

What is the problem?

Overview

With a bit of reflection, we can obtain

1. a uniform, and practical, syntax for both *records* (semantics) and *termtypes* (syntax)
2. on-the-fly unbundling; and,
3. mechanically obtain data structures from theories

Overview

With a bit of reflection, we can obtain

1. a uniform, and practical, syntax for both *records* (semantics) and *termtypes* (syntax)
2. on-the-fly unbundling; and,
3. mechanically obtain data structures from theories

'theory' τ	'data structure' `termtype` τ
pointed set	$\mathbb{1}$
dynamic system	$\mathbb{N}$
monoid	tree skeletons
collections	lists
graphs	(homogeneous) pairs
actions	infinite streams

What does a “module, package, context” look like?

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

Monoids model *unityped composition*: Sticking words on a page, sequencing programs, following instructions.

What is *in a* monoid?

People work with monoids at various levels of exposure ...

What is *in a* monoid?

People work with monoids at various levels of exposure . . .

- "Let M be a monoid, . . ."

What is *in a* monoid?

People work with monoids at various levels of exposure . . .

- “Let M be a monoid, . . .”
- “Given a monoid over $\mathbb{N}$, . . .”

What is *in a* monoid?

People work with monoids at various levels of exposure ...

- "Let M be a monoid, ..."
- "Given a monoid over $\mathbb{N}$, ..."
- "Consider *the* monoid $(\mathbb{N}, +)$, ..."
    - (Unique viz proof irrelevance.)

What is *in a* monoid?

People work with monoids at various levels of exposure . . .

- “Let M be a monoid, . . .”
- “Given a monoid over $\mathbb{N}$, . . .”
- “Consider *the* monoid $(\mathbb{N}, +)$, . . .”
  - (Unique viz proof irrelevance.)
- “Consider *the* monoid $(\mathbb{N}, +, 0)$, . . .”

“A monoid consists of a collection `Carrier`, an operation, …”?

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

Use-case: The category of monoids.

"A monoid over a given collection `Carrier` and operation `_⨾_` is given by ensuring there is a selected point ..."?

```
record Monoid₁
        (Carrier : Set)    : Set  where
  field -- ⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵ Change here
        _⨾_     :  Carrier → Carrier → Carrier
        Id      :  Carrier
        lid     : ∀ {x} →   Id ⨾ x  ≡  x
        rid     : ∀ {x} →   x ⨾ Id  ≡  x
        assoc   : ∀ {x y z} →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)
```

Use-case: Sharing the carrier type

Or . . . ?

```
record Monoid₂
        (Carrier : Set)
        (_⨾_     :  Carrier → Carrier → Carrier) : Set where
  field -- ⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵ Change here
        Id       :  Carrier
        lid      : ∀ {x} →    Id ⨾ x  ≡  x
        rid      : ∀ {x} →    x ⨾ Id  ≡  x
        assoc    : ∀ {x y z} →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)
```

Use-case: The additive monoid on the ℕatural numbers

Or . . . ?

```
record Monoid₃
        (Carrier : Set)
        (_⨾_     :  Carrier → Carrier → Carrier)
        (Id      :  Carrier)       : Set  where
  field -- ⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵⟵ Change here
        lid     : ∀ {x} →    Id ⨾ x  ≡  x
        rid     : ∀ {x} →    x ⨾ Id  ≡  x
        assoc   : ∀ {x y z} →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)
```

Tom Hales —Kepler Conjecture / Flyspeck

Structures are meaninglessly parameterized from a mathematical perspective. [...] That is, what is bundled cannot be later opened up as a parameter. [...] This means that library designers are forced to take a conservative approach and expose as a parameter anything that any user might reasonably want exposed, because once it is bundled, it is not coming back.

—*A Review of the Lean Theorem Prover*, *2018-09-18*

⇒ This is a problem we are solving!

Structures are meaninglessly parameterized from a mathematical perspective. [...] That is, what is bundled cannot be later opened up as a parameter. [...] This means that library designers are forced to take a conservative approach and expose as a parameter anything that any user might reasonably want exposed, because once it is bundled, it is not coming back.

—*A Review of the Lean Theorem Prover*, *2018-09-18*

$\Rightarrow$ This is a problem we are solving!

Structures are meaninglessly parameterized from a mathematical perspective. [...] That is, what is bundled cannot be later opened up as a parameter. [...] This means that library designers are forced to take a conservative approach and expose as a parameter anything that any user might reasonably want exposed, because once it is bundled, it is not coming back.

—*A Review of the Lean Theorem Prover*, *2018-09-18*

$\Rightarrow$ ***"The Unbundling Problem"***

Where does this *actually* happen?

- Agda's Standard Library,
- RATH-Agda,
- agda-categories
- Haskell's Standard Library

What are the adjacent problems?

Maintenance of relationships . . .

What are the adjacent problems?

Maintenance of relationships . . .

$$\texttt{Monoid}_0 \;\cong\; \Sigma\ \texttt{C : Set} \bullet \texttt{Monoid}_1\ \texttt{C}$$

What are the adjacent problems?

Maintenance of relationships ...

$$\texttt{Monoid}_0 \;\cong\; \Sigma\ \texttt{C} : \texttt{Set} \bullet \texttt{Monoid}_1\ \texttt{C}$$

$$\texttt{Monoid}_1\ \texttt{C} \;\cong\; \Sigma\ \mathsf{M} : \texttt{Monoid}_0 \bullet \texttt{Monoid}_0\texttt{.Carrier}\ \mathsf{M} \equiv \texttt{C}$$

What are the adjacent problems?

Maintenance of relationships . . .

$$\texttt{Monoid}_0 \;\cong\; \Sigma\; \texttt{C} : \texttt{Set} \bullet \texttt{Monoid}_1\; \texttt{C}$$

$$\texttt{Monoid}_1\; \texttt{C} \;\cong\; \Sigma\; \mathsf{M} : \texttt{Monoid}_0 \bullet \texttt{Monoid}_0\texttt{.Carrier}\; \mathsf{M} \equiv \texttt{C}$$

- Termtypes?
- Extensions?
- Exclusions?
- Pushouts: Name-relevant unions?

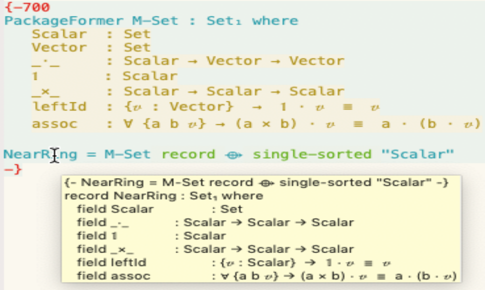
Roadmap —“PackageFormer ≈ Context ≈ JSON-Object”

1. The `PackageFormer` Prototype: A useful experimentation tool
2. The `Context` Library: Unbundling in Agda
3. Algebraic data types as a semantics for contexts

The `PackageFormer` Prototype: A useful experimentation tool

Evidence that the theory ‘actually works’

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



Generated code displayed on hover

But perhaps Haskell's type system does not give the programmer 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 $\texttt{Monoid}_0$, we will call Carrier "bundled up", while we call it "exposed" in $\texttt{Monoid}_1$ and $\texttt{Monoid}_2$. The bundled-up version allows us to speak of *a* monoid, rather than *a monoid on a given type* which is captured by $\texttt{Monoid}_1$. While $\texttt{Monoid}_2$ exposes both the carrier and the composition operation, we

automation, may want to use the associated datatype for 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 $\texttt{Monoid}_0$ 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.)

The Definition of a Monoid

```
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
```

The Definition of a Monoid

```
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
```

```
Monoid₀  = MonoidP record
Monoid₁  = Monoid₀ :waist 1
Monoid₂  = Monoid₀ :waist 2
Monoid₃  = Monoid₀ :waist 3
Monoid₃' = MonoidP record ⟴ unbundled 3
```

The Definition of a Monoid

```
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
```

```
Monoid₀  = MonoidP record
Monoid₁  = Monoid₀ :waist 1
Monoid₂  = Monoid₀ :waist 2
Monoid₃  = Monoid₀ :waist 3
Monoid₃' = MonoidP record ⟴ unbundled 3
```

```
Tree = MonoidP termtype-with-variables "Carrier"
≅
data Tree (Var : Set) : Set where
  inj : Var → Tree Var
  _⨾_   : Tree Var → Tree Var → Tree Var
  Id  : Tree Var
```

The Definition of a Monoid

```
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
```

```
Monoid₀  = MonoidP record
Monoid₁  = Monoid₀ :waist 1
Monoid₂  = Monoid₀ :waist 2
Monoid₃  = Monoid₀ :waist 3
Monoid₃' = MonoidP record ⟴ unbundled 3
```

```
 Tree = MonoidP termtype-with-variables "Carrier"
≅
 data Tree (Var : Set) : Set where
   inj : Var → Tree Var
   _⨾_   : Tree Var → Tree Var → Tree Var
   Id  : Tree Var
```

Linear effort in number of variations

The Definition of a Monoid

```
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
```

```
Monoid₀  = MonoidP record
Monoid₁  = Monoid₀ :waist 1
Monoid₂  = Monoid₀ :waist 2
Monoid₃  = Monoid₀ :waist 3
Monoid₃' = MonoidP record ⟴ unbundled 3
```

```
 Tree = MonoidP termtype-with-variables "Carrier"
≅
 data Tree (Var : Set) : Set where
   inj : Var → Tree Var
   _⨾_  : Tree Var → Tree Var → Tree Var
   Id  : Tree Var
```

Linear effort in number of variations

```
record : PackageFormer → PackageFormer
record = :kind record
         :alter-elements (λ es → (--map (map-qualifier (-const "field") it) es))
```

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

```
(𝒱 union pf (renaming₁ "") (renaming₂ "")
          (adjoin-retract₁ t) (adjoin-retract₂ t)
 = :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)))
   (-concat    ;; return value
      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)))))))
```

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

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
```

Primary Lessons Learned

1. Waist
2. Termtypes
3. Pragmatic

The Unbundling Problem —in Agda

What is “the” monoid on the natural numbers?

Some types can be viewed as a monoid in more than one way, e.g. both addition and multiplication on numbers. In such cases we often define newtypes and make those instances of Monoid, e.g. Sum and Product. —Hackage Data.Monoid

`Sum` $\alpha \cong \alpha$ `{- and -}` `Product` $\alpha \cong \alpha$

Alternate Solution to Multiple Monoid Instance Problem

Start with *fully bundled* `Monoid` then *expose fields as parameters* on the fly.

Alternate Solution to Multiple Monoid Instance Problem

Start with *fully bundled* `Monoid` then *expose fields as parameters* on the fly.

Reflection!

- Unfortunately, current mechanism cannot touch `record`-s *directly*.
- But every record is a Σ-type. . .

Records as $\Pi^w\Sigma$-types —Partitioned Contexts

- Instead of the nice *syntactic sugar*

```
record R (ε¹ : τ¹) ··· (εʷ : τʷ) : Set
  where
    field
      εʷ⁺¹ : τʷ⁺¹
      ⋮
      εʷ⁺ᵏ : τʷ⁺ᵏ
```

Records as $\Pi^w\Sigma$-types —Partitioned Contexts

- Instead of the nice *syntactic sugar*

```
record R (ε¹ : τ¹) ··· (εʷ : τʷ) : Set
  where
    field
      εʷ⁺¹ : τʷ⁺¹
      ⋮
      εʷ⁺ᵏ : τʷ⁺ᵏ
```

- Use a rawer form —*eek!*

```
R  : Π ε¹   : τ¹  • ··· • Π εʷ   : τʷ  • Set
R  ≅  λ ε¹   : τ¹  • ··· • λ εʷ   : τʷ
      • Σ εʷ⁺¹ : τʷ⁺¹ • ··· • Σ εʷ⁺ᵏ : τʷ⁺ᵏ
      • 𝟙
```

A Pragmatic Notation —Contexts

```
Monoid : Context ℓ₁
Monoid = do Carrier ← Set
            _⨾_      ← (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}
```

What is `Context`?

1. "Contexts" are exposure-indexed types

   ```
   Context = λ ℓ → (waist : ℕ) → Set ℓ
   ```

2. The "empty context" is the unit type

   ```
   End : ∀ {ℓ} → Context ℓ
   End {ℓ} = λ _ → 𝟙 {ℓ}
   ```

3. `do`-notation!

   ```
   _>>=_ : ∀ {a b}
         → (Γ : Context a)
         → (∀ {n} → Γ n → Context b)
         → Context (a ⊎ b)
   (Γ >>= f) zero    = Σ γ : Γ 0 • f γ 0
   (Γ >>= f) (suc n) = Π γ : Γ n • f γ n
   ```

Using Contexts —*reïfication*

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...
```

$$\texttt{Monoid} : \texttt{Context}$$

Using Contexts —*reïfication*

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...
```

$$\dfrac{\dfrac{\texttt{Monoid} : \texttt{Context}}{\texttt{Monoid 1} : \texttt{Set}}\text{[Application]}}{}$$

Using Contexts —*reïfication*

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...
```

$$\dfrac{\dfrac{\texttt{Monoid} \;:\; \texttt{Context}}{\texttt{Monoid 1} \;:\; \texttt{Set}}\,[\text{Application}]}{\texttt{Monoid 1}\ \mathbb{N} \;:\; \texttt{Set}}\,[\text{TypeError}]$$

Using Contexts —*reïfication*

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...
```

$$\dfrac{\dfrac{\texttt{Monoid : Context}}{\texttt{Monoid 1 : Set}}\text{[Application]}}{\texttt{Monoid 1 } \mathbb{N} \texttt{ : Set}}\text{[TypeError]}$$

$\Pi{\to}\lambda$ ‘‘Π^{w} x • τ’’ = ‘‘λ^{w} x • τ’’

Using Contexts —*reïfication*

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...
```

$$\dfrac{\dfrac{\texttt{Monoid : Context}}{\texttt{Monoid 1 : Set}}[\text{Application}]}{\texttt{Monoid 1 } \mathbb{N} \texttt{ : Set}}[\text{TypeError}]$$

```
Π→λ “Πʷ x • τ” = “λʷ x • τ”
```

```
C :waist w = Π→λ (C w)
```

Monoid$_i$

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...
```

Monoid$_i$

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...

Monoid :waist 0  : Set₁
Monoid :waist 0  ≡  Σ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • ...
```

Monoid$_i$

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...

Monoid :waist 0  : Set₁
Monoid :waist 0  ≡  Σ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • ...

Monoid :waist 1  :  Π C : Set • Set
Monoid :waist 1  =  λ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • ...
```

Monoid$_i$

```
Monoid : Context
Monoid = do C ← Set; _⨾_ : C → C → C; Id ← C; ...

Monoid :waist 0  : Set₁
Monoid :waist 0  ≡  Σ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • ...

Monoid :waist 1  :  Π C : Set • Set
Monoid :waist 1  =  λ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • ...

Monoid :waist 2  :  Π C : Set • Π _⨾_ : C → C → C • Set
Monoid :waist 2  =  λ C : Set • λ _⨾_ : C → C → C • Σ Id : C • ...
```

Example Instance —Additive Naturals

```
ℕ₊  : (Monoid ℓ₀ :waist 1) ℕ
ℕ₊  = ⟨ _+_              -- _⨾_
      , 0                -- Id
      , +-identityˡ
      , +-identityʳ
      , +-assoc
      ⟩
```

Summary: Solve the unbundling problem

‘Unbundle’ module fields as if they were parameters ‘on the fly’

Summary: Solve the unbundling problem

'Unbundle' module fields as if they were parameters 'on the fly'

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

Summary: Solve the unbundling problem

'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 = ⟨⟩
```

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

Datatypes for ASTs are also Contexts too!

From Contexts to Syntax Definitions

```
Monoid

⇝

do C ← Set; _⨾_ : C → C → C; Id : C; ...

⇝

λ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • ...

⇝

λ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • 𝟙

⇝

λ C : Set •       C × C    ⊎          C ⊎ 𝟙

⇝

μ C : Set •       C × C    ⊎          C ⊎ 𝟙
```

From Contexts to Syntax Definitions

```
termtype : UnaryFunctor → Type
termtype τ = Fix (Σ→⊎ (sources τ))
```

Monoid

⇝

do C ← Set; _⨾_ : C → C → C; Id : C; ...

⇝

λ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • ...

⇝

λ C : Set • Σ _⨾_ : C → C → C • Σ Id : C • 𝟙

⇝

λ C : Set • C × C ⊎ C ⊎ 𝟙

⇝

μ C : Set • C × C ⊎ C ⊎ 𝟙

Monoids give rise to tree skeletons / Context

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

Monoids give rise to tree skeletons / Termtype

```
𝕄 : Set
𝕄 = termtype (Monoid ℓ₀ :waist 1)

that-is : 𝕄
        ≡ Fix (λ X →
              -- _⊕_, branch
              X × X × 𝟙
              -- Id, nil leaf
           ⊎ 𝟙
              -- invariant leftId
           ⊎ 𝟘
              -- invariant rightId
           ⊎ 𝟘
              -- invariant assoc
           ⊎ 𝟘
              --  the “End {ℓ}”
           ⊎ 𝟘)
that-is = refl
```

Monoids give rise to tree skeletons / Readability

```
-- : 𝕄
pattern emptyM
    = μ (inj₂ (inj₁ tt))

-- : 𝕄 → 𝕄 → 𝕄
pattern branchM l r
    = μ (inj₁ (l , r , tt))

-- absurd 𝟘-values
pattern absurdM a
    = μ (inj₂ (inj₂ (inj₂ (inj₂ a))))
```

Monoids give rise to tree skeletons / termtype Monoid ≅ TreeSkeleton

```
data TreeSkeleton : Set where
  empty  : TreeSkeleton
  branch : TreeSkeleton → TreeSkeleton → TreeSkeleton
```

- "doing nothing"

```
to : 𝕄 → TreeSkeleton
to emptyM         = empty
to (branchM l r) = branch (to l) (to r)
to (absurdM (inj₁ ()))
to (absurdM (inj₂ ()))
```

- "doing nothing"

```
from : TreeSkeleton → 𝕄
from empty         = emptyM
from (branch l r) = branchM (from l) (from r)
```

Summary: Contexts $\mapsto$ {Records, Syntax}

Bring algebraic data types under the umbrella of grouping mechanisms:

Summary: Contexts ↦ {Records, Syntax}

Bring algebraic data types under the umbrella of grouping mechanisms:

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

Summary: Contexts ↦ {Records, Syntax}

Bring algebraic data types under the umbrella of grouping mechanisms:

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

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

Summary: Contexts ↦ {Records, Syntax}

Bring algebraic data types under the umbrella of grouping mechanisms:

```
DynamicSystem : Context ℓ₁                    data 𝔻 : Set where
DynamicSystem                                      startD : 𝔻
    = do State ← Set                               nextD  : 𝔻 → 𝔻
         start ← State
         next  ← (State → State)
         End
```

```
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
```

Summary: Common data-structures as free termtypes

'theory' τ	'data structure' $\texttt{termtype}\ \tau$
pointed set	$\mathbb{1}$
dynamic system	$\mathbb{N}$
monoid	tree skeletons
collections	lists
graphs	(homogeneous) pairs
actions	infinite streams

Many more theories τ to explore and see what data structures arise!

Conclusions

“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"

  `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`

Contributions

0. Identify the module design patterns used by DTL practitioners
1. Demonstrate that there is an expressive yet minimal set of primitives which allow common module constructions to be defined
2. Bring algebraic data types under the umbrella of grouping mechanisms
3. The ability to 'unbundle' module fields as if they were parameters 'on the fly'
4. Show that common data-structures are mechanically the (free) termtypes of common modules
5. Demonstrate that there is a practical implementation of such a framework
6. Finally, the resulting framework is *mostly* type-theory agnostic.

Contributions

0. Identify the module design patterns used by DTL practitioners
1. Demonstrate that there is an expressive yet minimal set of primitives which allow common module constructions to be defined
2. Bring algebraic data types under the umbrella of grouping mechanisms
3. The ability to 'unbundle' module fields as if they were parameters 'on the fly'
4. Show that common data-structures are mechanically the (free) termtypes of common modules
5. Demonstrate that there is a practical implementation of such a framework
6. Finally, the resulting framework is *mostly* type-theory agnostic.

$\Rightarrow$ Thank-you for your time! $\Leftarrow$