Do-it-yourself Module Systems

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

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

Overview

With a bit of reflection, we can obtain

1. a uniform, and practical, syntax for both *records* (semantics) and

The combinators presented in the thesis were guided *not* by theortetial concerns on the algebraic nature of containers but rather on the **practical needs of actual users working in DTLs**

2

3 …om theories

pointed set	$\mathbb{1}$
dynamic system	$\mathbb{N}$
monoid	tree skeletons
collections	lists
graphs	(homogeneous) pairs
actions	infinite streams

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
        _⨾_      :  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
        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
        lid     : ∀ {x} →    Id ⨾ x  ≡  x
        rid     : ∀ {x} →    x ⨾ Id  ≡  x
        assoc   : ∀ {x y z} →  (x ⨾ y) ⨾ z  ≡  x ⨾ (y ⨾ z)
```

Notice that the keyword `field` *is "going down" the waist each time.*

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*

⇒ This is a problem we are solving!

⇒ A *recent* problem

Structu[...]eterized 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!

$\Rightarrow$ A *recent* problem

$\Rightarrow$ **“The Unbundling Problem”**

Structu ... eterized from a mathematical perspec ... ter 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*

Does this *actually* happen?

Does this *actually* happen?

Yes!

Does this *actually* happen?

Yes!

Examples:

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

It's not just dependently-typed languages. . .

Does this *actually* happen?

Yes!

Examples:

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

It's not just dependently-typed languages. . .

- Arend
- Haskell's Standard Library

What are the adjacent problems?

Maintence of relationships when we "bring down the waist" —the position of the `field` keyword.

What are the adjacent problems?

Maintence of relationships when we "bring down the waist" —the position of the `field` keyword.

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

What are the adjacent problems?

Maintence of relationships when we "bring down the waist" —the position of the `field` keyword.

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

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

What are the adjacent problems?

Maintence of relationships when we "bring down the waist" —the position of the `field` keyword.

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

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

These coercions can be derived **mechanically**

What are the adjacent problems?

Maintence of relationships when we "bring down the waist" —the position of the `field` keyword.

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

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

These coercions can be derived **mechanically**

What about other *natural constructions* on mathematical theories (and the associated relationships)?

- Extensions? —"A group is a monoid with an extra..."

What are the adjacent problems?

Maintence of relationships when we “bring down the waist” —the position of the `field` keyword.

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

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

These coercions can be derived **mechanically**

What about other *natural constructions* on mathematical theories (and the associated relationships)?

- Extensions? —“A group is a monoid with an extra…”
- Exclusions? —“A semigroup is a non-unital monoid.”

What are the adjacent problems?

Maintence of relationships when we "bring down the waist" —the position of the `field` keyword.

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

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

These coercions can be derived **mechanically**

What about other *natural constructions* on mathematical theories (and the associated relationships)?

- Extensions? —"A group is a monoid with an extra..."
- Exclusions? —"A semigroup is a non-unital monoid."
- Termtypes? —"Lists are just the free *monoid over* a given type."

What are the adjacent problems?

Maintence of relationships when we "bring down the waist" —the position of the `field` keyword.

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

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

These coercions can be derived **mechanically**

What about other *natural constructions* on mathematical theories (and the associated relationships)?

- Extensions? —"A group is a monoid with an extra..."
- Exclusions? —"A semigroup is a non-unital monoid."
- Termtypes? —"Lists are just the free *monoid over* a given type."
- Pushouts: Name-relevant unions? —"A monoid is a pointed set along with a semigroup such that they share the same carrier."

What are the adjacent problems?

Maintence of relationships when we “bring down the waist” —the position of the `field` keyword.

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

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

These coercions can be derived **mechanically**

What about other *natural constructions* on mathematical theories (and the associated relationships)?

- Extensions? —“A group is a monoid with an extra…”
- Exclusions? —“A semigroup is a non-unital monoid.”
- Termtypes? —“Lists are just the free *monoid over* a given type.”
- Pushouts: Name-relevant unions? —“A monoid is a pointed set along with a semigroup such that they share the same carrier.”
- Numerous other constructions from Category Theory

Which items should be fields, which parameters?

- The `Monoid`$_i$ family showed some combinations of items selected as parameters.

Which items should be fields, which parameters?

- The `Monoid`$_i$ family showed some combinations of items selected as parameters.
- There are other combinations of what is to be exposed and hidden, for applications that we might never think of.

Which items should be fields, which parameters?

- The `Monoid`$_i$ family showed some combinations of items selected as parameters.
- There are other combinations of what is to be exposed and hidden, for applications that we might never think of.
- Providing always the most-general parameterisation produces awkward library interfaces!

Which items should be fields, which parameters?

- The `Monoid`$_i$ family showed some combinations of items selected as parameters.
- There are other combinations of what is to be exposed and hidden, for applications that we might never think of.
- Providing always the most-general parameterisation produces awkward library interfaces!

Proposed Solution:

- Commit to no particular formulation and allow on-the-fly "unbundling"
  - This is the *converse* of instantiation

Which items should be fields, which parameters?

- The `Monoid`$_i$ family showed some combinations of items selected as parameters.
- There are other combinations of what is to be exposed and hidden, for applications that we might never think of.
- Providing always the most-general parameterisation produces awkward library interfaces!

Proposed Solution:

- Commit to no particular formulation and allow on-the-fly "unbundling"
  - This is the *converse* of instantiation
- The "Emacs editor tactic" `PackageFormer`
- The "Agda library" `Context`

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!

```
{-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)
```

Generated code displayed on hover

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

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 Monoid$_0$, we will call Carrier "bundled up", while we call it "exposed" in Monoid$_1$ and 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 Monoid$_1$. While 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 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.)

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

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}
```

⇒ Influenced Agda's Standard Library

```
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 Monoid$_0$, we will call Carrier "bundled up", while we call it "exposed" in Monoid$_1$ and 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 Monoid$_1$. While 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 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

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
```

We regain the different candidates by applying variationals.

```
Monoid₀  =  MonoidP record
Monoid₁  =  MonoidP record ⟴ unbundled 1
Monoid₂  =  MonoidP record ⟴ unbundled 2
Monoid₃  =  Monoid₀’ exposing "Carrier; _⨾_; Id"
```

. . . and we can do more

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
```

We regain the different candidates by applying variationals.

```
Monoid₀  =  MonoidP record
Monoid₁  =  MonoidP record ⟴ unbundled 1
Monoid₂  =  MonoidP record ⟴ unbundled 2
Monoid₃  =  Monoid₀′ exposing "Carrier; _⨾_; Id"
```

…and we can do more

Monoid syntax!

```
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
```

We regain the different candidates by applying variationals.

```
Monoid₀  =  MonoidP record
Monoid₁  =  MonoidP record ⟴ unbundled 1
Monoid₂  =  MonoidP record ⟴ unbundled 2
Monoid₃  =  Monoid₀′ exposing "Carrier; _⨾_; Id"
```

...and we can do more

Monoid syntax!

```
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

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)))))))
```

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

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 →
```

Framework built around 5 metaprimitives
↦ Lisp Metaprogramming, untyped string manipulation,
↦ Macro DSL, Agda generation

```
     (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)))))))
```

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

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 →
```

Framework built around 5 metaprimitives

↦ Lisp Metaprogramming, untyped string manipulation,

↦ Macro DSL, Agda generation

```
        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"
```

⇒ The rest are “user-defined” with a bit of Lisp

```
   (-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)))))))
```

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
```

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

$\mapsto$ Useful, typecheckable, dauntingly large code

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

Terse, readable, specifications

$\mapsto$ Useful, typecheckable, dauntingly large code

200+ **one-line** specs

$\mapsto$ 1500+ lines of typechecked Agda

```
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
```

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
```

Terse, readable, specifications

↦ Useful, typecheckable, dauntingly large code

```
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
```

200+ **one-line** specs

↦ 1500+ lines of typechecked Agda

⇒ 750% efficiency savings

```
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                                                ⟴ :remark "over Magma"
PointedSteinerMagm                                    :remark "over Magma"
UnipotentPointedMagma    = PointedMagma extended-by’ "unipotent  : ∀ (x : U) →  (x * x) ≡ e"
Sloop                    = PointedSteinerMagma union’ UnipotentPointedMagma
```

Useful engineering result

Primary Lessons Learned

Primary Lessons 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 Lessons 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.

Termtypes Record types (Σ), type classes ($\Pi^1\Sigma$), and algebraic data types ($\mathcal{W}$) are all valid semantics of contexts —which are "name : type = optional-definition" tuples.

Primary Lessons 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.

Termtypes Record types (Σ), type classes ($\Pi^1\Sigma$), and algebraic data types ($\mathcal{W}$) are all valid semantics of contexts —which are "name : type = optional-definition" tuples.

Pragmatic We have an extendable,

Primary Lessons 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.

Termtypes Record types (Σ), type classes ($\Pi^1\Sigma$), and algebraic data types ($\mathcal{W}$) are all valid semantics of contexts —which are "name : type = optional-definition" tuples.

Pragmatic We have an extendable, expressive,

Primary Lessons 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.

Termtypes Record types (Σ), type classes ($\Pi^1\Sigma$), and algebraic data types ($\mathcal{W}$) are all valid semantics of contexts —which are "name : type = optional-definition" tuples.

Pragmatic We have an extendable, expressive, and efficient interface based on a small kernel,

Primary Lessons 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.

Termtypes Record types (Σ), type classes ($\Pi^1\Sigma$), and algebraic data types ($\mathcal{W}$) are all valid semantics of contexts —which are "name : type = optional-definition" tuples.

Pragmatic We have an extendable, expressive, and efficient interface based on a small kernel, that is immediately usable,

Primary Lessons 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.

Termtypes Record types (Σ), type classes ($\Pi^1\Sigma$), and algebraic data types ($\mathcal{W}$) are all valid semantics of contexts —which are "name : type = optional-definition" tuples.

Pragmatic We have an extendable, expressive, and efficient interface based on a small kernel, that is immediately usable, as an editor extension;

Primary Lessons 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.

Termtypes Record types (Σ), type classes ($\Pi^1\Sigma$), and algebraic data types ($\mathcal{W}$) are all valid semantics of contexts —which are "name : type = optional-definition" tuples.

Pragmatic We have an extendable, expressive, and efficient interface based on a small kernel, that is immediately usable, as an editor extension; what about an in-language (DTL) library?

The Unbundling Problem —in Agda

What is “the” monoid on the natural numbers?

Haskell’s solution is to make two isomorphic copies of numbers since typeclass instance search relies on *unique* instances for the typeclass parameters.

> *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*

$$\texttt{Sum}\ \alpha \cong \alpha \quad \textit{\{- and -\}}\ \texttt{Product}\ \alpha \cong \alpha$$

For `Num` α they have different monoid instances.

Alternate Solution to Multiple Monoid Instance Problem

Alternate Solution to Multiple Monoid Instance Problem

Start with *fully bundled* `Monoid`

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.

How?

Alternate Solution to Multiple Monoid Instance Problem

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

How?

Reflection!

Alternate Solution to Multiple Monoid Instance Problem

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

How?

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*

  $\texttt{record R } (\varepsilon^1 : \tau^1) \cdots (\varepsilon^w : \tau^w) : \texttt{Set}$

  $\quad\texttt{where}$

  $\quad\quad\texttt{field}$

  $\quad\quad\quad\varepsilon^{w+1} : \tau^{w+1}$

  $\quad\quad\quad\vdots$

  $\quad\quad\quad\varepsilon^{w+k} : \tau^{w+k}$

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

- Instead of the nice *syntactic sugar*

$$
\begin{array}{l}
\texttt{record R } (\varepsilon^1 : \tau^1) \cdots (\varepsilon^w : \tau^w) : \texttt{Set} \\
\quad \texttt{where} \\
\qquad \texttt{field} \\
\qquad\quad \varepsilon^{w+1} : \tau^{w+1} \\
\qquad\quad \vdots \\
\qquad\quad \varepsilon^{w+k} : \tau^{w+k}
\end{array}
$$

- Use a more raw form —*eek!*

$$
\begin{array}{l}
\texttt{R} : \Pi\ \varepsilon^1 : \tau^1 \bullet \cdots \bullet \Pi\ \varepsilon^w : \tau^w \bullet \texttt{Set} \\
\texttt{R} \cong \lambda\ \varepsilon^1 : \tau^1 \bullet \cdots \bullet \lambda\ \varepsilon^w : \tau^w \\
\qquad \bullet\ \Sigma\ \varepsilon^{w+1} : \tau^{w+1} \bullet \cdots \bullet \Sigma\ \varepsilon^{w+k} : \tau^{w+k} \\
\qquad \bullet\ \mathbb{1}
\end{array}
$$

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 more raw 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
```

$\Leftarrow$ "parameters"

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

- Instead of the nice *syntactic sugar*

$$\texttt{record R } (\varepsilon^1 : \tau^1) \cdots (\varepsilon^w : \tau^w) : \texttt{Set}$$
$$\quad \texttt{where}$$
$$\qquad \texttt{field}$$
$$\qquad\quad \varepsilon^{w+1} : \tau^{w+1}$$
$$\qquad\quad \vdots$$
$$\qquad\quad \varepsilon^{w+k} : \tau^{w+k}$$

- Use a more raw form —*eek!*

$$\begin{array}{ll} \texttt{R} & : \Pi\ \varepsilon^1 : \tau^1 \bullet \cdots \bullet \Pi\ \varepsilon^w : \tau^w \bullet \texttt{Set} \\ \texttt{R} & \cong\ \lambda\ \varepsilon^1 : \tau^1 \bullet \cdots \bullet \lambda\ \varepsilon^w : \tau^w \\ & \quad \bullet\ \Sigma\ \varepsilon^{w+1} : \tau^{w+1} \bullet \cdots \bullet \Sigma\ \varepsilon^{w+k} : \tau^{w+k} \\ & \quad \bullet\ \mathbb{1} \end{array}$$

$\Leftarrow$ "parameters"
$\Leftarrow$ "fields"

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

- Instead of the nice *syntactic sugar*

$$
\begin{array}{l}
\mathtt{record}\ \mathtt{R}\ (\varepsilon^1 : \tau^1) \cdots (\varepsilon^w : \tau^w) : \mathtt{Set} \\
\quad \mathtt{where} \\
\quad\quad \mathtt{field} \\
\quad\quad\quad \varepsilon^{w+1} : \tau^{w+1} \\
\quad\quad\quad \vdots \\
\quad\quad\quad \varepsilon^{w+k} : \tau^{w+k}
\end{array}
$$

- Use a more raw form —*eek!*

$$
\begin{array}{lll}
\mathtt{R} & : \Pi\ \varepsilon^1 : \tau^1 \bullet \cdots \bullet \Pi\ \varepsilon^w : \tau^w \bullet \mathtt{Set} & \Leftarrow \text{“parameters”} \\
\mathtt{R} & \cong \lambda\ \varepsilon^1 : \tau^1 \bullet \cdots \bullet \lambda\ \varepsilon^w : \tau^w & \\
& \quad \bullet\ \Sigma\ \varepsilon^{w+1} : \tau^{w+1} \bullet \cdots \bullet \Sigma\ \varepsilon^{w+k} : \tau^{w+k} & \Leftarrow \text{“fields”} \\
& \quad \bullet\ \mathbb{1} &
\end{array}
$$

We say w is the **“waist”**

A Pragmatic Notation —Contexts

1. "Contexts" are exposure-indexed types

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

A Pragmatic Notation —Contexts

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}
   ```

A Pragmatic Notation —Contexts

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
```

A Pragmatic Notation —Contexts

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
```

The "DIY" lies at »=, permitting Σ, Π, $\mathcal{W}$, `let`, . . . !

Example Context —Monoids

```
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}
```

Using Contexts —*reïfication*

- If `C : Context` ℓ_0 then `C w` has the type Π^w `x` $\bullet$ τ —consisting of *w*-many Π's—

Using Contexts —*reïfication*

- If `C : Context` ℓ_0 then `C w` has the type Π^w `x` $\bullet$ τ —consisting of *w*-many Π's— but we want to apply `C w` to *w*-many *parameters*. . .

Using Contexts —*reïfication*

- If `C : Context` ℓ_0 then `C w` has the type Π^w `x` $\bullet$ τ —consisting of *w*-many Π's— but we want to apply `C w` to *w*-many *parameters*. . .
- So we need a combinator. . .

  $\Pi{\rightarrow}\lambda$ “Π^w `x` $\bullet$ τ” = “λ^w `x` $\bullet$ τ”

Using Contexts —*reïfication*

- If `C : Context` ℓ_0 then `C w` has the type Π^w `x` $\bullet$ τ —consisting of w-many Π's— but we want to apply `C w` to w-many *parameters*. . .

- So we need a combinator. . .

  $\Pi{\rightarrow}\lambda$ ‘‘Π^w `x` $\bullet$ τ’’ `=` ‘‘λ^w `x` $\bullet$ τ’’

- with an infix form for contexts in particular . . .

  `C :waist w` `=` $\Pi{\rightarrow}\lambda$ `(C w)`

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

$$\begin{aligned} \Pi{\rightarrow}\lambda\ (\Pi\ a : A \bullet \tau) &= (\lambda\ a : A \bullet \tau) \\ C\ \texttt{:waist}\ w &= \Pi{\rightarrow}\lambda\ (C\ w) \end{aligned}$$

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

$$\Pi{\rightarrow}\lambda\ (\Pi\ \mathtt{a} : \mathtt{A} \bullet \tau) = (\lambda\ \mathtt{a} : \mathtt{A} \bullet \tau)$$
$$\mathtt{C}\ \mathtt{:waist}\ \mathtt{w} = \Pi{\rightarrow}\lambda\ (\mathtt{C}\ \mathtt{w})$$

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

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

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

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

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

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

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

```
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
```

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

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

```
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*

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; ...
```

With no parameters, we have a $\Pi^0\Sigma$-type (a record)

```
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; ...
```

With no parameters, we have a $\Pi^0\Sigma$-type (a record)

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

With one parameter, we have a typeclass

```
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; ...
```

With no parameters, we have a $\Pi^0\Sigma$-type (a record)

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

With one parameter, we have a typeclass

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

With two parameters, we have a *'solution'* to the additive-or-multiplicative-monoid-problem!

```
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
      ⟩
```

Lessons Learned

On-the-fly unbundling can be implemented as an in-language library in a dependently-typed language with sufficient reflection capabilities :-)

⋆ ⋆ ⋆

The `Context` approach *inherits* the strengths and limitations of the host language.

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

GADTs are Contexts too!

From Contexts to GADTS

`Monoid`

From Contexts to GADTS

`Monoid`

$\rightsquigarrow$

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

From Contexts to GADTS

Monoid

$\rightsquigarrow$

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

$\rightsquigarrow$

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

From Contexts to GADTS

Monoid

$\rightsquigarrow$

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

$\rightsquigarrow$

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

$\rightsquigarrow$

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

From Contexts to GADTS

Monoid

$\rightsquigarrow$

$\texttt{do}\ \texttt{C} \leftarrow \texttt{Set};\ \_⨾\_ : \texttt{C} \to \texttt{C} \to \texttt{C};\ \texttt{Id} : \texttt{C};\ \ldots$

$\rightsquigarrow$

$\lambda\ \texttt{C} : \texttt{Set} \bullet \Sigma\ \_⨾\_ : \texttt{C} \to \texttt{C} \to \texttt{C} \bullet \Sigma\ \texttt{Id} : \texttt{C} \bullet \ldots$

$\rightsquigarrow$

$\lambda\ \texttt{C} : \texttt{Set} \bullet \Sigma\ \_⨾\_ : \texttt{C} \to \texttt{C} \to \texttt{C} \bullet \Sigma\ \texttt{Id} : \texttt{C} \bullet \mathbb{1}$

$\rightsquigarrow$

$\lambda\ \texttt{C} : \texttt{Set} \bullet \quad \texttt{C} \times \texttt{C} \quad \uplus \quad \texttt{C} \uplus \mathbb{1}$

From Contexts to GADTS

Monoid

$\rightsquigarrow$

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

$\rightsquigarrow$

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

$\rightsquigarrow$

λ C : Set •

termtype : UnaryFunctor → Type

termtype τ = Fix (Σ→⊎ (sources τ))

$\rightsquigarrow$

λ C : Set • C × C ⊎ C ⊎ $\mathbb{1}$

$\rightsquigarrow$

μ C : Set • C × C ⊎ C ⊎ $\mathbb{1}$

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

'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!

Contributions

Module Systems for DTLs

0. Identify the module design patterns used by DTL practitioners

Module Systems for DTLs

0. Identify the module design patterns used by DTL practitioners
1. The ability to *implement* module systems for DTLs within DTLs

Module Systems for DTLs

0. Identify the module design patterns used by DTL practitioners
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

0. Identify the module design patterns used by DTL practitioners
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

0. Identify the module design patterns used by DTL practitioners
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 = ⟨⟩
```

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

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.
   - ☒ There are various forms of semantics presented in the thesis: Abstract semantics via signatures, concrete semantics via Agda functions, denotational semantics via $\Pi\Sigma\mathcal{W}$, as well as a guide for forming the `Context` library in other languages.

"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`

“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`

“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`

“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`

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