Functional Pearl: Do-it-yourself module types

ANONYMOUS AUTHOR(S)

Can parameterised records and algebraic datatypes be derived from one pragmatic declaration?

Record types give a universe of discourse, parameterised record types fix parts of that universe ahead of time, and algebraic datatypes give us first-class syntax, whence evaluators and optimisers.

The answer is in the affirmative. Besides a practical shared declaration interface, which is extensible in the language, we also find that common data structures correspond to simple theories.

1 INTRODUCTION

All too often, when we program, we write the same information two or more times in our code, in different guises. For example, in Haskell, we may write a class, a record to reify that class, and an algebraic type to give us a syntax for programs written using that class. In proof assistants, this tends to get worse rather than better, as parametrized records give us a means to "stage" information. From here on, we will use Agda [Norell 2007] for our examples.

Concretely, suppose we have two monoids (M_1, $\_⨾_1\_$, Id_1) and (M_2, $\_⨾_2\_$, Id_2), if we know[1] that `ceq : M₁ ≡ M₂` then it is "obvious" that $Id_2 ⨾_2 (x ⨾_1 Id_1) \equiv x$ for all $x : M_1$. However, as written, this does not type-check. This is because $\_⨾_2\_$ expects elements of M_2 but has been given an element of M_1. Because we have `ceq` in hand, we can use `subst` to transport things around. The resulting formula, shown as the type of `claim` below, then typechecks, but is hideous. "subst hell" only gets worse. Below, we use pointed magmas for brevity, as the problem is the same.

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

module Awkward-Formulation (A B : Magma₀)
    (ceq : Magma₀.Carrier A ≡ Magma₀.Carrier B)
    where
      open Magma₀ A renaming (Id to Id₁; _⨾_ to _⨾₁_)
      open Magma₀ B renaming (Id to Id₂; _⨾_ to _⨾₂_)

      claim : ∀ x → Id₂ ⨾₂ subst id ceq (x ⨾₁ Id₁) ≡ subst id ceq x
      claim = {!!}
      {- "{!!}" stands for a "hole" in Agda,
         needing replacement by an expression -}
```

It should not be this difficult to state a trivial fact. We could make things artifically prettier by defining coe to be `subst id ceq` without changing the heart of the matter. But if $Magma_0$ is the definition used in the library we are using, we are stuck with it, if we want to be compatible with other work.

[1] The propositional equality $M_1 \equiv M_2$ means the M_i are convertible with each other when all free variables occurring in the M_i are instantiated, and otherwise are not necessarily identical. A stronger equality operator cannot be expressed in Agda.

2018. 2475-1421/2018/1-ART $15.00
https://doi.org/

Ideally, we would prefer to be able to express that the carriers are shared "on the nose", which can be done as follows:

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

module Nicer
    (M : Set)    {- The shared carrier -}
    (A B : Magma₁ M)
    where
      open Magma₁ A renaming (Id to Id₁; _⨾_ to _⨾₁_)
      open Magma₁ B renaming (Id to Id₂; _⨾_ to _⨾₂_)

      claim : ∀ x → Id₂ ⨾₂ (x ⨾₁ Id₁) ≡ x
      claim = {!!}
```

This is the formulation we expected, without noise. Thus it seems that it would be better to expose the carrier. But, before long, we'd find a different concept, such as homomorphism, which is awkward in this way, and cleaner using the first approach. These two approaches are called *bundled* and *unbundled* respectively [Spitters and van der Weegen 2011].

The definitions of homomorphism themselves (see below) is not so different, but the definition of composition already starts to be quite unwieldly.

```
record Hom₀ (A B : Magma₀) : Set where ···
record Hom₁ {M₁ M₂ : Set} (A : Magma₁ M₁) (B : Magma₁ M₂) : Set where ···

composition₀ : ∀ {A B C} → Hom₀ A B → Hom₀ B C → Hom₀ A C
composition₀ = {!!}

composition₁ : ∀ {M₁ M₂ M₃} {A : Magma₁ M₁} {B : Magma₁ M₂} {C : Magma₁ M₃}
               → Hom₁ A B → Hom₁ B C → Hom₁ A C
composition₁ = {!!}
```

So not only are there no general rules for when to bundle or not, it is in fact guaranteed that any given choice will be sub-optimal for certain applications. Furthermore, these types are equivalent, as we can "pack away" an exposed piece, e.g., `Monoid₀ ≅ Σ M : Set • Monoid₁ M`. The developers of the Agda standard library [agd 2020] have chosen to expose all types and function symbols while bundling up the proof obligations at one level, and also provide a fully bundled form as a wrapper. This is also the method chosen in Lean [Hales 2018], and in Coq [Spitters and van der Weegen 2011].

While such a choice is workable, it is still not optimal. There are bundling variants that are unavailable, and would be more convenient for certain applications.

We will show an automatic technique for unbundling data at will; thereby resulting in *bundling-independent representations* and in *delayed unbundling*. Our contributions are to show:

(1) Languages with sufficiently powerful type systems and meta-programming can conflate record and term datatype declarations into one practical interface. In addition, the contents of these grouping mechanisms may be function symbols as well as propositional invariants —an example is shown at the end of Section 3. We identify the problem and the subtleties in shifting between representations in Section 2.

(2) Parameterised records can be obtained on-demand from non-parameterised records (Section 3) .
  - As with $\texttt{Magma}_0$, the traditional approach [Gross et al. 2014] to unbundling a record requires the use of transport along propositional equalities, with trivial `refl`-exivity proofs. In Section 3, we develop a combinator, `_:waist_`, which removes the boilerplate necessary at the type specialisation location as well as at the instance declaration location.

(3) Programming with fixed-points of unary type constructors can be made as simple as programming with term datatypes (Section 4).

(4) Astonishingly, we mechanically regain ubiquitous data structures such as `ℕ, Maybe, List` as the term datatypes of simple pointed and monoidal theories (Section 5).

As an application, in Section 6 we show that the resulting setup applies as a semantics for a declarative pre-processing tool that accomplishes the above tasks.

For brevity, and accessibility, a number of definitions are elided and only dashed pseudo-code is presented in the paper, with the understanding that such functions need to be extended homomorphically over all possible term constructors of the host language. Enough is shown to communicate the techniques and ideas, as well as to make the resulting library usable. The details, which users do not need to bother with, can be found in the appendices.

2 THE PROBLEMS

There are a number of problems, with the number of parameters being exposed being the pivotal concern. To exemplify the distinctions at the type level as more parameters are exposed, consider the following approaches to formalising a dynamical system —a collection of states, a designated start state, and a transition function.

```
record DynamicSystem₀ : Set₁ where
  field
    State : Set
    start  : State
    next   : State → State

record DynamicSystem₁ (State : Set) : Set where
  field
    start : State
    next  : State → State

record DynamicSystem₂ (State : Set) (start : State) : Set where
  field
    next : State → State
```

Each $\texttt{DynamicSystem}_i$ is a type constructor of `i`-many arguments; but it is the types of these constructors that provide insight into the sort of data they contain:

Type	Kind
$\texttt{DynamicSystem}_0$	$\texttt{Set}_1$
$\texttt{DynamicSystem}_1$	`Π X : Set • Set`
$\texttt{DynamicSystem}_2$	`Π X : Set • Π x : X • Set`

We shall refer to the concern of moving from a record to a parameterised record as **the unbundling problem** [Garillot et al. 2009]. For example, moving from the *type* $\texttt{Set}_1$ to the *function type* `Π X : Set • Set` gets us from $\texttt{DynamicSystem}_0$ to something resembling $\texttt{DynamicSystem}_1$, which we arrive at if we can obtain a *type constructor* `λ X : Set • ⋯`. We shall refer to the latter change as *reïfication* since the result is more concrete: It can be applied. This transformation will be denoted by Π→λ. To clarify this subtlety, consider the following forms of the polymorphic identity function. Notice that id_i *exposes* `i`-many details at the type level to indicate the sort of data it consists of. However, notice that id_0 is a type of functions whereas id_1 is a function on types. Indeed, the latter two are derived from the first one: $\mathrm{id}_{i+1} = \Pi{\to}\lambda\ \mathrm{id}_i$ These identities are true by `refl`-exivity —see Appendix A.8.

```
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
```

Of course, there is also the need for descriptions of values, which leads to term datatypes. We shall refer to the shift from record types to algebraic data types as **the termtype problem**. Our aim is to obtain all of these notions —of ways to group data together— from a single user-friendly context declaration, using monadic notation.

3 MONADIC NOTATION

There is little use in an idea that is difficult to use in practice. As such, we conflate records and termtypes by starting with an ideal syntax they would share, then derive the necessary artefacts that permit it. Our choice of syntax is monadic do-notation [Marlow et al. 2016; Moggi 1991]:

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

Here `Context`, `End`, and the underlying monadic bind operator are unknown. Since we want to be able to *expose* a number of fields at will, we may take `Context` to be types indexed by a number denoting exposure. Moreover, since records are product types, we expect there to be a recursive definition whose base case will be the identity of products, the unit type 𝟙 —which corresponds to ⊤ in the Agda standard library and to `()` in Haskell.

Exposure	Elaboration
0	Σ State : Set • Σ start : X • Σ next : State → State • 𝟙
1	Π State : Set • Σ start : X • Σ next : State → State • 𝟙
2	Π State : Set • Π start : X • Σ next : State → State • 𝟙
3	Π State : Set • Π start : X • Π next : State → State • 𝟙

Table 1. Elaborations of DynamicSystem at various exposure levels

With these elaborations of `DynamicSystem` to guide the way, we resolve two of our unknowns.

```
{- “Contexts” are exposure-indexed types -}
Context = λ ℓ → ℕ → Set ℓ
```

```
{- Every type can be used as a context -}
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S

{- The “empty context” is the unit type -}
End : ∀ {ℓ} → Context ℓ
End = ‘ 𝟙
```

It remains to identify the definition of the underlying bind operation >>=. Usually, for a type constructor `m`, bind is typed `∀ {X Y : Set} → m X → (X → m Y) → m Y`. It allows one to “extract an `X`-value for later use” in the `m Y` context. Since our `m = Context` is from levels to types, we need to slightly alter bind’s typing.

```
_>>=_ : ∀ {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 definition here accounts for the current exposure index: If zero, we have *record types*, otherwise *function types*. Using this definition, the above dynamical system context would need to be expressed using the lifting quote operation.

```
‘ Set >>= λ State → ‘ State >>= λ start → ‘ (State → State) >>= λ next → End
{- or -}
do State ← ‘ Set
   start ← ‘ State
   next  ← ‘ (State → State)
   End
```

Interestingly [Bird 2009; Hudak et al. 2007], use of do-notation in preference to bind, >>=, was suggested by John Launchbury in 1993 and was first implemented by Mark Jones in Gofer. Anyhow, with our goal of practicality in mind, we shall “build the lifting quote into the definition” of bind:

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

Listing 1. Semantics: Context do-syntax is interpreted as Π-Σ-types

With this definition, the above declaration `DynamicSystem` typechecks. However, `DynamicSystem` i $\ncong$ `DynamicSystem`$_i$, instead DynamicSystem i are “factories”: Given `i`-many arguments, a product value is formed. What if we want to *instantiate* some of the factory arguments ahead of time?

```
𝒩₀ : DynamicSystem 0    {- See the elaborations in Table 1 -}
𝒩₀ = ℕ , 0 , suc , tt
```

```
𝒩₁ : DynamicSystem 1
𝒩₁ = λ State → ??? {- Impossible to complete if “State” is empty! -}

{- “Instantiaing” X to be ℕ in “DynamicSystem 1” -}
𝒩₁’ : let State = ℕ in Σ start : State  • Σ s : (State → State)  • 𝟙
𝒩₁’ = 0 , suc , tt
```

It seems what we need is a method, say Π→λ, that takes a Π-type and transforms it into a λ-expression. One could use a universe, an algebraic type of codes denoting types, to define Π→λ. However, one can no longer then easily use existing types since they are not formed from the universe's constructors, thereby resulting in duplication of existing types via the universe encoding. This is neither practical nor pragmatic.

As such, we are left with pattern matching on the language's type formation primitives as the only reasonable approach. The method Π→λ is thus a macro[2] that acts on the syntactic term representations of types. Below is main transformation —the details can be found in Appendix A.7.

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

That is, we walk along the term tree replacing occurrences of Π with λ. For example,

```
  Π→λ (Π→λ (DynamicSystem 2))
≡{- Definition of DynamicSystem at exposure level 2 -}
  Π→λ (Π→λ (Π X : Set • Π s : X  • Σ n : X → X  • 𝟙))
≡{- Definition of Π→λ -}
  Π→λ (λ X : Set • Π s : X  • Σ n : X → X  • 𝟙)
≡{- Homomorphy of Π→λ -}
  λ X : Set • Π→λ (Π s : X  • Σ n : X → X  • 𝟙)
≡{- Definition of Π→λ -}
  λ X : Set • λ s : X  • Σ n : X → X  • 𝟙
```

For practicality, `_:waist_` is a macro (defined in Appendix A.9) acting on contexts that repeats Π→λ a number of times in order to lift a number of field components to the parameter level.

```
τ :waist n = Π→λⁿ (τ n)
f⁰ x       = x
fⁿ⁺¹ x     = fⁿ (f x)
```

We can now “fix arguments ahead of time”. Before such demonstration, we need to be mindful of our practicality goals: One declares a grouping mechanism with `do ... End`, which in turn has its instance values constructed with `⟨ ... ⟩`.

```
-- Expressions of the form “⋯ , tt” may now be written “⟨ ⋯ ⟩”
infixr 5 ⟨ _⟩
⟨⟩ : ∀ {ℓ} → 𝟙 {ℓ}
⟨⟩ = tt

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s
```

[2] A *macro* is a function that manipulates the abstract syntax trees of the host language. In particular, it may take an arbitrary term, shuffle its syntax to provide possibly meaningless terms or terms that could not be formed without pattern matching on the possible syntactic constructions. An up to date and gentle introduction to reflection in Agda can be found at [Al-hassy 2019b]

```
_⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × (𝟙 {ℓ})
s ⟩ = s , tt
```

The following instances of grouping types demonstrate how information moves from the body level to the parameter level.

```
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 = ⟨⟩
```

Using `:waist` i we may fix the first i-parameters ahead of time. Indeed, the type `(DynamicSystem :waist 1) ℕ` is *the type of dynamic systems over carrier* ℕ, whereas `(DynamicSystem :waist 2) ℕ 0` is *the type of dynamic systems over carrier* ℕ *and start state 0*.

Examples of the need for such on-the-fly unbundling can be found in numerous places in the Haskell standard library. For instance, the standard libraries [dat 2020] have two isomorphic copies of the integers, called `Sum` and `Product`, whose reason for being is to distinguish two common monoids: The former is for *integers with addition* whereas the latter is for *integers with multiplication*. An orthogonal solution would be to use contexts:

```
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}
```

With this context, `(Monoid` ℓ_0 `:waist 2) M ⊕` is the type of monoids over *particular* types `M` and *particular* operations ⊕. Of-course, this is orthogonal, since traditionally unification on the carrier type `M` is what makes typeclasses and canonical structures [Mahboubi and Tassi 2013] useful for ad-hoc polymorphism.

4 TERMTYPES AS FIXED-POINTS

We have a practical monadic syntax for possibly parameterised record types that we would like to extend to termtypes. Algebraic data types are a means to declare concrete representations of the least fixed-point of a functor; see [Swierstra 2008] for more on this idea. In particular, the description language 𝔻 for dynamical systems, below, declares concrete constructors for a fixpoint of a certain functor `F`; i.e., `𝔻 ≅ Fix F` where:

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

```
F : Set → Set
F = λ (D : Set) → 𝟙 ⊎ D

data Fix (F : Set → Set) : Set where
  μ : F (Fix F) → Fix F
```

The problem is whether we can derive `F` from `DynamicSystem`. Let us attempt a quick calculation sketching the necessary transformation steps (informally expressed via "⇒"):

```
  do S ← Set; s ← S; n ← (S → S); End
⇒ {- Use existing interpretation to obtain a record. -}
  Σ S : Set • Σ s : S • Σ n : (S → S) • 𝟙
⇒ {- Pull out the carrier, ":waist 1",
    to obtain a type constructor using "Π→λ". -}
  λ S : Set • Σ s : S • Σ n : (S → S) • 𝟙
⇒ {- Termtype constructors target the declared type,
    so only their sources matter. E.g., 's : S' is a
    nullary constructor targeting the carrier 'S'.
    This introduces 𝟙 types, so any existing
    occurances are dropped via 𝟘. -}
  λ S : Set • Σ s : 𝟙 • Σ n : S • 𝟘
⇒ {- Termtypes are sums of products. -}
  λ S : Set •       𝟙   ⊎     S  ⊎ 𝟘
⇒ {- Termtypes are fixpoints of type constructors. -}
  Fix (λ X • 𝟙 ⊎ S)  -- i.e., 𝔻
```

Since we may view an algebraic data-type as a fixed-point of the functor obtained from the union of the sources of its constructors, it suffices to treat the fields of a record as constructors, then obtain their sources, then union them. That is, since algebraic-datatype constructors necessarily target the declared type, they are determined by their sources. For example, considered as a unary constructor `op : A → B` targets the termtype `B` and so its source is `A`. The details on the operations ⇊, Σ→⊎, and sources characterised by the pseudocode below can be found in appendices A.3.4, A.12.4, and A.12.3, respectively. It suffices to know that Σ→⊎ rewrites dependent-sums into disjoint sums, which requires the second argument to lose its reference to the first argument which is accomplished by ⇊; further details can be found in the appendices.

```
⇊ τ = "reduce all de Bruijn indices within τ by 1"

Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (⇊ Ba)

sources (λ x : (Π a : A • Ba) • τ) = (λ x : A • sources τ)
sources (λ x : A              • τ) = (λ x : 𝟙 • sources τ)

termtype τ = Fix (Σ→⊎ (sources τ))
```

It is instructive to work through the process of how 𝔻 is obtained from `termtype` in order to demonstrate that this approach to algebraic data types is practical.

```
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
```

With these **pattern** declarations, we can actually use the more meaningful names `startD` and `nextD` when pattern matching, instead of the seemingly daunting μ-`inj`-ections. For instance, we can immediately see that the natural numbers act as the description language for dynamical systems:

```
to : 𝔻 → ℕ
to startD     = 0
to (nextD x) = suc (to x)

from : ℕ → 𝔻
from zero     = startD
from (suc n) = nextD (from n)
```

Readers whose language does not have **pattern** clauses need not despair. With the macro

```
Inj n x = μ (inj₂ ⁿ (inj₁ x))
```

we may define `startD = Inj 0 tt` and `nextD e = Inj 1 e` —that is, constructors of termtypes are particular injections into the possible summands that the termtype consists of. Details on this macro may be found in appendix A.12.6.

5 FREE DATATYPES FROM THEORIES

Astonishingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if a parameterised context `C : Set → Context ℓ₀` is given, then

```
ℂ = λ X → termtype (C X :waist 1)
```

can be used to form 'free, lawless, C-instances'. For instance, earlier we witnessed that the termtype of dynamical systems is essentially the natural numbers.

Theory	Termtype
Dynamical Systems	ℕ
Pointed Structures	Maybe
Monoids	Binary Trees

Table 2. Data structures as free theories

The final entry in Table 2 is a well known correspondence that we can now not only formally express, but also prove to be true.

```
𝕄 : Set
𝕄 = termtype (Monoid ℓ₀ :waist 1)
{- i.e., Fix (λ X → 𝟙       -- Id, nil leaf
               ⊎ X × X × 𝟙 -- _⊕_, branch
               ⊎ 𝟘         -- invariant leftId
               ⊎ 𝟘         -- invariant rightId
               ⊎ X × X × 𝟘 -- invariant assoc
               ⊎ 𝟘)        -- the "End {ℓ}"
-}

-- Pattern synonyms for more compact presentation
pattern emptyM      = μ (inj₂ (inj₁ tt))             -- : 𝕄
```

```
pattern branchM l r = μ (inj₁ (l , r , tt))            -- : 𝕄 → 𝕄 → 𝕄
pattern absurdM a   = μ (inj₂ (inj₂ (inj₂ (inj₂ a))))  -- absurd values of 𝟘

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

Using Agda's Emacs interface, we may interactively case-split on values of 𝕄 until the declared patterns appear, then we associate them with the constructors of `TreeSkeleton`.

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

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

That these two operations are inverses is easily demonstrated.

```
from∘to : ∀ m → from (to m) ≡ m
from∘to emptyM        = refl
from∘to (branchM l r) = cong₂ branchM (from∘to l) (from∘to r)
from∘to (absurdM (inj₁ ()))
from∘to (absurdM (inj₂ ()))

to∘from : ∀ t → to (from t) ≡ t
to∘from empty         = refl
to∘from (branch l r) = cong₂ branch (to∘from l) (to∘from r)
```

Without the **`pattern`** declarations the result would remain true, but it would be quite difficult to believe in the correspondence without a machine-checked proof.

To obtain a data structure over some 'value type' Ξ, one must start with "theories containing a given set Ξ". For example, we could begin with the theory of abstract collections, then obtain lists as the associated termtype.

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

ℂ : Set → Set
ℂ Elem = termtype ((Collection ℓ₀ :waist 2) Elem)

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

```
      to : ∀ {E} → ℂ E → List E
      to (e :: es) = e :: to es
      to ∅         = []
```

It is then little trouble to show that `to` is invertible. We invite the readers to join in on the fun and try it out themselves!

6 RELATED WORKS

Surprisingly, conflating parameterised and non-parameterised record types with termtypes *within a language in a practical fashion* has not been done before.

The PackageFormer [Al-hassy 2019a; Al-hassy et al. 2019] editor extension reads contexts —in nearly the same notation as ours— enclosed in dedicated comments, then generates and imports Agda code from them seamlessly in the background whenever typechecking happens. The framework provides a fixed number of meta-primitives for producing arbitrary notions of grouping mechanisms, and allows arbitrary Emacs Lisp [Graham 1995] to be invoked in the construction of complex grouping mechanisms.

	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

Table 3. Comparing the in-language Context mechanism with the PackageFormer editor extension

The PackageFormer paper [Al-hassy et al. 2019] provided the syntax necessary to form useful grouping mechanisms but was shy on the semantics of such constructs. We have chosen the names of our combinators to closely match those of PackageFormer's with an aim of furnishing the mechanism with semantics by construing the syntax as semantics-functions; i.e., we have a shallow embedding of PackageFormer's constructs as Agda entities:

Syntax	Semantics
`PackageFormer`	`Context`
`:waist`	`:waist`
⟴	Forward function application
`:kind`	`:kind`, see below
`:level`	Agda built-in
`:alter-elements`	Agda macros

Table 4. Contexts as a semantics for PackageFormer constructs

PackageFormer's `_:kind_` meta-primitive dictates how an abstract grouping mechanism should be viewed in terms of existing Agda syntax. However, unlike PackageFormer, all of our syntax consists of legitimate Agda terms. Since language syntax is being manipulated, we are forced to implement the `_:kind_` meta-primitive as a macro —further details can be found in Appendix A.13.

```
data Kind : Set where
  ‘record    : Kind
  ‘typeclass : Kind
  ‘data      : Kind
```

```
𝒞 :kind ‘record = 𝒞 0
𝒞 :kind ‘typeclass = 𝒞 :waist 1
𝒞 :kind ‘data = termtype (𝒞 :waist 1)
```

We did not expect to be able to define a full Agda implementation of the semantics of PackageFormer's syntactic constructs due to Agda's rather constrained metaprogramming mechanism. However, it is important to note that PackageFormer's Lisp extensibility expedites the process of trying out arbitrary grouping mechanisms —such as partial-choices of pushouts and pullbacks along user-provided assignment functions— since it is all either string or symbolic list manipulation. On the Agda side, using contexts, it would require substantially more effort due to the limited reflection mechanism and the intrusion of the stringent type system.

7 CONCLUSION

Starting from the insight that related grouping mechanisms could be unified, we showed how related structures can be obtained from a single declaration using a practical interface. The resulting framework, based on contexts, still captures the familiar record declaration syntax as well as the expressivity of usual algebraic datatype declarations —at the minimal cost of using **`pattern`** declarations to aide as user-chosen constructor names. We believe that our approach to using contexts as general grouping mechanisms *with* a practical interface are interesting contributions.

We used the focus on practicality to guide the design of our context interface, and provided interpretations both for the rather intuitive "contexts are name-type records" view, and for the novel "contexts are fixed-points" view for termtypes. In addition, to obtain parameterised variants, we needed to explicitly form "contexts whose contents are over a given ambient context" —e.g., contexts of vector spaces are usually discussed with the understanding that there is a context of fields that can be referenced— which we did using the name binding machanism of do-notation. These relationships are summarised in the following table.

Concept	Concrete Syntax	Description
Context	`do S ← Set; s ← S; n ← (S → S); End`	"name-type pairs"
Record Type	`Σ S : Set • Σ s : S • Σ n : S → S • 𝟙`	"bundled-up data"
Function Type	`Π S • Σ s : S • Σ n : S → S • 𝟙`	"a type of functions"
Type constructor	`λ S • Σ s : S • Σ n : S → S • 𝟙`	"a function on types"
Algebraic datatype	`data 𝔻 : Set where s : 𝔻; n : 𝔻 → 𝔻`	"a descriptive syntax"

Table 5. Contexts embody all kinds of grouping mechanisms

To those interested in exotic ways to group data together —such as, mechanically deriving product types and homomorphism types of theories— we offer an interface that is extensible using Agda's reflection mechanism. In comparison with, for example, special-purpose preprocessing tools, this has obvious advantages in accessibility and semantics.

To Agda programmers, this offers a standard interface for grouping mechanisms that had been sorely missing, with an interface that is so familiar that there would be little barrier to its use. In particular, as we have shown, it acts as an in-language library for exploiting relationships between free theories and data structures. As we have only presented the high-level definitions of the core combinators, leaving the Agda-specific details to the appendices, it is also straightforward to translate the library into other dependently-typed languages.

REFERENCES

2020. Agda Standard Library. https://github.com/agda/agda-stdlib

2020. Haskell Basic Libraries — Data.Monoid. http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Monoid.html

Musa Al-hassy. 2019a. The Next 700 Module Systems: Extending Dependently-Typed Languages to Implement Module System Features In The Core Language. https://alhassy.github.io/next-700-module-systems-proposal/thesis-proposal.pdf

Musa Al-hassy. 2019b. A slow-paced introduction to reflection in Agda —Tactics! https://github.com/alhassy/gentle-intro-to-reflection

Musa Al-hassy, Jacques Carette, and Wolfram Kahl. 2019. A language feature to unbundle data at will (short paper). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*, Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm (Eds.). ACM, 14–19. https://doi.org/10.1145/3357765.3359523

Richard Bird. 2009. Thinking Functionally with Haskell. (2009). https://doi.org/10.1017/cbo9781316092415

François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics (Lecture Notes in Computer Science)*, Tobias Nipkow and Christian Urban (Eds.), Vol. 5674. Springer, Munich, Germany. https://hal.inria.fr/inria-00368403

Paul Graham. 1995. *ANSI Common Lisp.* Prentice Hall Press, USA.

Jason Gross, Adam Chlipala, and David I. Spivak. 2014. Experience Implementing a Performant Category-Theory Library in Coq. arXiv:math.CT/1401.7694v2

Tom Hales. 2018. A Review of the Lean Theorem Prover. https://jiggerwit.wordpress.com/2018/09/18/a-review-of-the-lean-theorem-prover/

Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, Barbara G. Ryder and Brent Hailpern (Eds.). ACM, 1–55. https://doi.org/10.1145/1238844.1238856

Assia Mahboubi and Enrico Tassi. 2013. Canonical Structures for the working Coq user. In *ITP 2013, 4th Conference on Interactive Theorem Proving (LNCS)*, Sandrine Blazy, Christine Paulin, and David Pichardie (Eds.), Vol. 7998. Springer, Rennes, France, 19–34. https://doi.org/10.1007/978-3-642-39634-2_5

Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 92–104. https://doi.org/10.1145/2976002.2976007

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory.* Ph.D. Dissertation. Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology.

Bas Spitters and Eelis van der Weegen. 2011. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science* 21, 4 (2011), 795–825. https://doi.org/10.1017/S0960129511000119

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Jim Woodcock and Jim Davies. 1996. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., USA.

A APPENDICES

Below is the entirety of the Context library discussed in the paper proper.

```
-- Agda version 2.6.0.1
-- Standard library version 1.2

module Context where
```

Also included are unit tests, evidence for claims made in the paper proper, and a brief case-study on graphs to demonstrate some features of the Context library that are necessary for practical use, such as field projections, but which did not receive attention in the paper proper.

A.1 Imports

```
open import Level renaming (_⊔_ to _⊎_; suc to ℓsuc; zero to ℓ₀)
open import Relation.Binary.PropositionalEquality
open import Relation.Nullary

open import Data.Nat
open import Data.Fin  as Fin using (Fin)
open import Data.Maybe  hiding (_>>=_)

open import Data.Bool using (Bool ; true ; false)
open import Data.List as List using (List ; [] ; _::_ ; _::ʳ_; sum)

ℓ₁   = Level.suc ℓ₀
```

A.2 Quantifiers Π:•/Σ:• and Products/Sums

We shall using Z-style quantifier notation [Woodcock and Davies 1996] in which the quantifier dummy variables are separated from the body by a large bullet.

In Agda, we use `\:` to obtain the "ghost colon" since standard colon `:` is an Agda operator.

Even though Agda provides `∀ (x : τ) → fx` as a built-in syntax for Π-types, we have chosen the Z-style one below to mirror the notation for Σ-types, which Agda provides as `record` declarations. In the paper proper, in the definition of bind, the subtle shift between Σ-types and Π-types is easier to notice when the notations are so similar that only the quantifier symbol changes.

```
open import Data.Empty using (⊥)
open import Data.Sum
open import Data.Product
open import Function using (_∘_)

Σ:• : ∀ {a b} (A : Set a) (B : A → Set b) → Set _
Σ:• = Σ

infix -666 Σ:•
syntax Σ:• A (λ x → B) = Σ x : A • B

Π:• : ∀ {a b} (A : Set a) (B : A → Set b) → Set _
Π:• A B = (x : A) → B x

infix -666 Π:•
syntax Π:• A (λ x → B) = Π x : A • B

record ⊤ {ℓ} : Set ℓ where
  constructor tt

𝟙 = ⊤ {ℓ₀}
𝟘 = ⊥
```

A.3 Reflection

We form a few metaprogramming utilities we would have expected to be in the standard library.

```
import Data.Unit as Unit
open import Reflection hiding (name; Type) renaming (_>>=_ to _>>=ₘ_)
```

Before continuing, there are a few difficulties about Agda's metaprogramming capabilities that should be mentioned:

(1) Even when recursion is on structurally smaller terms of abstract syntax trees, termination cannot be automatically deduced. As such, we request Agda to believe us that certain definitions are terminating.
(2) Since Agda macros cannot be recursive —possibly due to issues of termination— an idiom we use to define a recursive operation on terms then wrap that in Agda's typechecking monad to form macros.
(3) Sometimes, no matter how explicit we make certain affairs, macro invocations will complain about being unable to infer certain details. As a workaround, we type any declaration involving a macro invocation before using it —inference is difficult in dependently-typed settings and even worse in the presence of metaprogramming.

A.3.1 *Single argument application.*

```
_app_ : Term → Term → Term
(def f args) app arg' = def f (args ::^r arg (arg-info visible relevant) arg')
(con f args) app arg' = con f (args ::^r arg (arg-info visible relevant) arg')
{-# CATCHALL #-}
tm app arg' = tm
```

Notice that we maintain existing applications:

```
quoteTerm (f x) app quoteTerm y  ≈  quoteTerm (f x y)
```

A.3.2 *Reify ℕ term encodings as ℕ values.*

```
toℕ : Term → ℕ
toℕ (lit (nat n)) = n
{-# CATCHALL #-}
toℕ _ = 0
```

A.3.3 *The Length of a Term.*

```
arg-term : ∀ {ℓ} {A : Set ℓ} → (Term → A) → Arg Term → A
arg-term f (arg i x) = f x

{-# TERMINATING #-}
length_t : Term → ℕ
length_t (var x args)      = 1 + sum (List.map (arg-term length_t ) args)
length_t (con c args)      = 1 + sum (List.map (arg-term length_t ) args)
length_t (def f args)      = 1 + sum (List.map (arg-term length_t ) args)
length_t (lam v (abs s x)) = 1 + length_t x
length_t (pat-lam cs args) = 1 + sum (List.map (arg-term length_t ) args)
length_t (Π[ x : A ] Bx)   = 1 + length_t Bx
{-# CATCHALL #-}
-- sort, lit, meta, unknown
length_t t = 0
```

Here is an example use:

```
_ : length_t (quoteTerm (Σ x : ℕ • x ≡ x)) ≡ 10
_ = refl
```

A.3.4 *Decreasing de Brujin Indices.*

Given a quantification (⊕ x : τ • fx), its body fx may refer to a free variable x. If we decrement all de Bruijn indices fx contains, then there would be no reference to x.

```
var-dec_0 : (fuel : ℕ) → Term → Term
var-dec_0 zero t  = t
-- Let's use an “impossible” term.
var-dec_0 (suc n) (var zero args)      = def (quote ⊥) []
var-dec_0 (suc n) (var (suc x) args)   = var x args
```

```
var-dec₀ (suc n) (con c args)          = con c (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (def f args)          = def f (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (lam v (abs s x))     = lam v (abs s (var-dec₀ n x))
var-dec₀ (suc n) (pat-lam cs args)     = pat-lam cs (map-Args (var-dec₀ n) args)
var-dec₀ (suc n) (Π[ s : arg i A ] B) = Π[ s : arg i (var-dec₀ n A) ] var-dec₀ n B
{-# CATCHALL #-}
-- sort, lit, meta, unknown
var-dec₀ n t = t
```

In the paper proper, var-dec was mentioned once under the name ⇊.

```
var-dec : Term → Term
var-dec t = var-dec₀ (lengthₜ t) t
```

Notice that we made the decision that x, the body of (⊕ x • x), will reduce to 𝟘, the empty type. Indeed, in such a situation the only Debrujin index cannot be reduced further. Here is an example:

```
_ : ∀ {x : ℕ} → var-dec (quoteTerm x) ≡ quoteTerm ⊥
_ = refl
```

A.4 Context Monad

```
Context = λ ℓ → ℕ → Set ℓ

infix -1000 ‘_
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S

End : ∀ {ℓ} → Context ℓ
End = ‘ ⊤

End₀ = End {ℓ₀}

_>>=_ : ∀ {a b}
      → (Γ : Set a)  -- Main diference
      → (Γ → Context b)
      → Context (a ⊍ b)
(Γ >>= f) ℕ.zero  = Σ γ : Γ • f γ 0
(Γ >>= f) (suc n) = (γ : Γ) → f γ n
```

A.5 ⟨⟩ Notation

```
-- Expressions of the form “⋯ , tt” may now be written “⟨ ⋯ ⟩”
infixr 5 ⟨ _⟩
⟨⟩ : ∀ {ℓ} → ⊤ {ℓ}
⟨⟩ = tt

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

_⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × ⊤ {ℓ}
s ⟩ = s , tt
```

A.6 DynamicSystem Context

```
DynamicSystem : Context (ℓsuc Level.zero)
DynamicSystem = do X ← Set
                   z ← X
                   s ← (X → X)
                   End {Level.zero}
```

```
-- Records with n-Parameters, n : 0..3
A B C D : Set₁
A = DynamicSystem 0 -- Σ X : Set  • Σ z : X  • Σ s : X → X  • ⊤
B = DynamicSystem 1 --  (X : Set) → Σ z : X  • Σ s : X → X  • ⊤
C = DynamicSystem 2 --  (X : Set)    (z : X) → Σ s : X → X  • ⊤
D = DynamicSystem 3 --  (X : Set)    (z : X) →  (s : X → X) → ⊤

_ : A ≡ (Σ X : Set  • Σ z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : B ≡ (Π X : Set  • Σ z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : C ≡ (Π X : Set  • Π z : X  • Σ s : (X → X)  • ⊤) ; _ = refl
_ : D ≡ (Π X : Set  • Π z : X  • Π s : (X → X)  • ⊤) ; _ = refl

stability : ∀ {n} →   DynamicSystem (3 + n)
                    ≡ DynamicSystem  3
stability = refl

B-is-empty : ¬ B
B-is-empty b = proj₁( b ⊥)

𝒩₀ : DynamicSystem 0
𝒩₀ = ℕ , 0 , suc , tt

𝒩 : DynamicSystem 0
𝒩 = ⟨ ℕ , 0 , suc ⟩

B-on-ℕ : Set
B-on-ℕ = let X = ℕ in Σ z : X  • Σ s : (X → X)  • ⊤

ex : B-on-ℕ
ex = ⟨ 0 , suc ⟩
```

A.7 Π→λ

```
Π→λ-helper : Term → Term
Π→λ-helper (pi  a b)         = lam visible b
Π→λ-helper (lam a (abs x y)) = lam a (abs x (Π→λ-helper y))
{-# CATCHALL #-}
Π→λ-helper x = x

macro
  Π→λ : Term → Term → TC Unit.⊤
  Π→λ tm goal = normalise tm >>=ₘ λ tm' → unify (Π→λ-helper tm') goal
```

A.8 $\mathbf{id}_{i+1} \approx$ Π→λ $\mathbf{id}_i$

```
_ : id₁ ≡ Π→λ id₀
_ = refl

_ : id₂ ≡ Π→λ id₁
_ = refl
```

A.9 `_:waist_`

```
waist-helper : ℕ → Term → Term
waist-helper zero t    = t
waist-helper (suc n) t = waist-helper n (Π→λ-helper t)

macro
  _:waist_ : Term → Term → Term → TC Unit.⊤
```

```
  _:waist_ t n goal =      normalise (t app n)
                       >>=m λ t' → unify (waist-helper (toℕ n) t') goal
```

A.10 DynamicSystem :waist *i*

```
A' : Set₁
B' : ∀ (X : Set) → Set
C' : ∀ (X : Set) (x : X) → Set
D' : ∀ (X : Set) (x : X) (s : X → X) → Set

A' = DynamicSystem :waist 0
B' = DynamicSystem :waist 1
C' = DynamicSystem :waist 2
D' = DynamicSystem :waist 3

𝒩⁰ : A'
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : B' ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : C' ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : D' ℕ 0 suc
𝒩³ = ⟨⟩
```

It may be the case that Γ `0` ≡ Γ `:waist 0` for every context Γ.

```
_ : DynamicSystem 0 ≡ DynamicSystem :waist 0
_ = refl
```

A.11 Field projections

```
Field₀ : ℕ → Term → Term
Field₀ zero c    = def (quote proj₁) (arg (arg-info visible relevant) c ∷ [])
Field₀ (suc n) c = Field₀ n (def (quote proj₂) (arg (arg-info visible relevant) c ∷ []))

macro
  Field : ℕ → Term → Term → TC Unit.⊤
  Field n t goal = unify goal (Field₀ n t)
```

An example usage can be found below in the setting of graphs.

A.12 Termtypes

Using the guiding calculation outlined in the paper proper we shall form $\mathtt{D}_i$ for each stage in the calculation.

A.12.1 Stage 1: Records.

```
D₁ = DynamicSystem 0

1-records : D₁ ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • ⊤)
1-records = refl
```

A.12.2 Stage 2: Parameterised Records.

```
D₂ = DynamicSystem :waist 1

2-funcs : D₂ ≡ (λ (X : Set) → Σ z : X • Σ s : (X → X) • ⊤)
2-funcs = refl
```

A.12.3 Stage 3: Sources. Let's begin with an example to motivate the definition of sources.

```
_ :   quoteTerm (∀ {x : ℕ} → ℕ)
    ≡ pi (arg (arg-info hidden relevant) (quoteTerm ℕ)) (abs "x" (quoteTerm ℕ))
_ = refl
```

We now form two sources-helper utilities, although we suspect they could be combined into one function.

```
sources₀ : Term → Term
-- Otherwise:
sources₀ (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) =
    def (quote _×_) (vArg A
                    :: vArg (def (quote _×_)
                                 (vArg (var-dec Ba)
                                      :: vArg (var-dec (var-dec (sources₀ Cab))) :: []))
                    :: [])
sources₀ (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 𝟘
sources₀ (Π[ x : arg i A ] Bx) = A
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources₀ t = quoteTerm 𝟙

{-# TERMINATING #-}
sources₁ : Term → Term
sources₁ (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 𝟘
sources₁ (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) = def (quote _×_) (vArg A ::
  vArg (def (quote _×_) (vArg (var-dec Ba)
                             :: vArg (var-dec (var-dec (sources₀ Cab))) :: [])) :: [])
sources₁ (Π[ x : arg i A ] Bx) = A
sources₁ (def (quote Σ) (ℓ₁ :: ℓ₂ :: τ :: body))
    = def (quote Σ) (ℓ₁ :: ℓ₂ :: map-Arg sources₀ τ :: List.map (map-Arg sources₁) body)
-- This function introduces 𝟙s, so let's drop any old occurances a la 𝟘.
sources₁ (def (quote ⊤) _) = def (quote 𝟘) []
sources₁ (lam v (abs s x))     = lam v (abs s (sources₁ x))
sources₁ (var x args) = var x (List.map (map-Arg sources₁) args)
sources₁ (con c args) = con c (List.map (map-Arg sources₁) args)
sources₁ (def f args) = def f (List.map (map-Arg sources₁) args)
sources₁ (pat-lam cs args) = pat-lam cs (List.map (map-Arg sources₁) args)
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources₁ t = t
```

We now form the macro and some unit tests.

```
macro
  sources : Term → Term → TC Unit.⊤
  sources tm goal = normalise tm >>=ₘ λ tm' → unify (sources₁ tm') goal

_ : sources (ℕ → Set) ≡ ℕ
_ = refl

_ : sources (Σ x : (ℕ → Fin 3) • ℕ) ≡ (Σ x : ℕ • ℕ)
_ = refl

_ : ∀ {ℓ : Level} {A B C : Set}
  → sources (Σ x : (A → B) • C) ≡ (Σ x : A • C)
_ = refl

_ : sources (Fin 1 → Fin 2 → Fin 3) ≡ (Σ _ : Fin 1 • Fin 2 × 𝟙)
```

```
_ = refl

_ : sources (Σ f : (Fin 1 → Fin 2 → Fin 3 → Fin 4) • Fin 5)
  ≡ (Σ f : (Fin 1 × Fin 2 × Fin 3) • Fin 5)
_ = refl

_ : ∀ {A B C : Set} → sources (A → B → C) ≡ (A × B × 𝟙)
_ = refl

_ : ∀ {A B C D E : Set} → sources (A → B → C → D → E)
                          ≡ Σ A (λ _ → Σ B (λ _ → Σ C (λ _ → Σ D (λ _ → ⊤))))
_ = refl
```

Design decision: Types starting with implicit arguments are *invariants*, not *constructors*.

```
-- one implicit
_ : sources (∀ {x : ℕ} → x ≡ x) ≡ 𝟘
_ = refl

-- multiple implicits
_ : sources (∀ {x y z : ℕ} → x ≡ y) ≡ 𝟘
_ = refl
```

The third stage can now be formed.

```
D₃ = sources D₂

3-sources : D₃ ≡ λ (X : Set) → Σ z : 𝟙 • Σ s : X • 𝟘
3-sources = refl
```

A.12.4 Stage 4: Σ→⊎ −Replacing Products with Sums.

```
{-# TERMINATING #-}
Σ→⊎₀ : Term → Term
Σ→⊎₀ (def (quote Σ) (h₁ :: h₀ :: arg i A :: arg i₁ (lam v (abs s x)) :: []))
  =  def (quote _⊎_) (h₁ :: h₀ :: arg i A :: vArg (Σ→⊎₀ (var-dec x)) :: [])
-- Interpret “End” in do-notation to be an empty, impossible, constructor.
Σ→⊎₀ (def (quote ⊤) _) = def (quote ⊥) []
 -- Walk under λ's and Π's.
Σ→⊎₀ (lam v (abs s x)) = lam v (abs s (Σ→⊎₀ x))
Σ→⊎₀ (Π[ x : A ] Bx) = Π[ x : A ] Σ→⊎₀ Bx
{-# CATCHALL #-}
Σ→⊎₀ t = t

macro
  Σ→⊎ : Term → Term → TC Unit.⊤
  Σ→⊎ tm goal = normalise tm >>=ₘ λ tm’ → unify (Σ→⊎₀ tm’) goal
```

Unit tests:

```
_ : Σ→⊎ (Π X : Set • (X → X))       ≡ (Π X : Set • (X → X)); _ = refl
_ : Σ→⊎ (Π X : Set • Σ s : X • X) ≡ (Π X : Set • X ⊎ X)  ; _ = refl
_ : Σ→⊎ (Π X : Set • Σ s : (X → X) • X) ≡ (Π X : Set • (X → X) ⊎ X)  ; _ = refl
_ : Σ→⊎ (Π X : Set • Σ z : X • Σ s : (X → X) • ⊤ {ℓ₀}) ≡ (Π X : Set • X ⊎ (X → X) ⊎ ⊥)
_ = refl


D₄ = Σ→⊎ D₃

4-unions : D₄ ≡ λ X → 𝟙 ⊎ X ⊎ 𝟘
4-unions = refl
```

A.12.5 Stage 5: Fixpoint and proof that $\mathbb{D} \cong \mathbb{N}$. Since we want to define algebraic data-types as fixed-points, we are led inexorably to using a recursive type that fails to be positive.

```
{-# NO_POSITIVITY_CHECK #-}
data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
  μ : F (Fix F) → Fix F
module termtype[DynamicSystem]≅ℕ where

  𝔻 = Fix D₄

  -- Pattern synonyms for more compact presentation
  pattern zeroD   = μ (inj₁ tt)        -- : 𝔻
  pattern sucD e = μ (inj₂ (inj₁ e)) -- : 𝔻 → 𝔻

  to : 𝔻 → ℕ
  to zeroD    = 0
  to (sucD x) = suc (to x)

  from : ℕ → 𝔻
  from zero    = zeroD
  from (suc n) = sucD (from n)

  to∘from : ∀ n → to (from n) ≡ n
  to∘from zero    = refl
  to∘from (suc n) = cong suc (to∘from n)

  from∘to : ∀ d → from (to d) ≡ d
  from∘to zeroD    = refl
  from∘to (sucD x) = cong sucD (from∘to x)
```

A.12.6 `termtype` *and* `Inj` *macros.* We summarise the stages together into one macro: "`termtype : UnaryFunctor → Type`".

```
macro
  termtype : Term → Term → TC Unit.⊤
  termtype tm goal =
                normalise tm
          >>=ₘ λ tm' → unify goal (def (quote Fix) ((vArg (Σ→⊎₀ (sources₁ tm'))) :: []))
```

It is interesting to note that in place of `pattern` clauses, say for languages that do not support them, we would resort to "fancy injections".

```
Inj₀ : ℕ → Term → Term
Inj₀ zero c    = con (quote inj₁) (arg (arg-info visible relevant) c :: [])
Inj₀ (suc n) c = con (quote inj₂) (vArg (Inj₀ n c) :: [])

-- Duality!
-- i-th projection: proj₁ ∘ (proj₂ ∘ ··· ∘ proj₂)
-- i-th injection:  (inj₂ ∘ ··· ∘ inj₂) ∘ inj₁

macro
  Inj : ℕ → Term → Term → TC Unit.⊤
  Inj n t goal = unify goal ((con (quote μ) []) app (Inj₀ n t))
```

With this alternative, we regain the "user chosen constructor names" for $\mathbb{D}$:

```
startD : 𝔻
startD = Inj 0 (tt {ℓ₀})

nextD' : 𝔻 → 𝔻
nextD' d = Inj 1 d
```

A.13 The _:kind_ meta-primitive

```
data Kind : Set where
  ‘record    : Kind
  ‘typeclass : Kind
  ‘data      : Kind

macro
  _:kind_ : Term → Term → Term → TC Unit.⊤
  _:kind_ t (con (quote ‘record) _)    goal = normalise (t app (quoteTerm 0))
                     >>=_m λ t’ → unify (waist-helper 0 t’) goal
  _:kind_ t (con (quote ‘typeclass) _) goal = normalise (t app (quoteTerm 1))
                     >>=_m λ t’ → unify (waist-helper 1 t’) goal
  _:kind_ t (con (quote ‘data) _) goal = normalise (t app (quoteTerm 1))
                     >>=_m λ t’ → normalise (waist-helper 1 t’)
                     >>=_m λ t’’ → unify goal (def (quote Fix)
                                              ((vArg (Σ→⊎_0 (sources_1 t’’))) :: []))
  _:kind_ t _ goal = unify t goal
```

Informally, _:kind_ behaves as follows:

```
C :kind ‘record    = C :waist 0
C :kind ‘typeclass = C :waist 1
C :kind ‘data      = termtype (C :waist 1)
```

A.14 Example: Graphs in Two Ways

There are two ways to implement the type of graphs in the dependently-typed language Agda: Having the vertices be a parameter or having them be a field of the record. Then there is also the syntax for graph vertex relationships. Suppose a library designer decides to work with fully bundled graphs, $\texttt{Graph}_0$ below, then a user decides to write the function comap, which relabels the vertices of a graph, using a function f to transform vertices.

```
record Graph_0 : Set_1 where
  constructor ⟨_,_⟩_0
  field
    Vertex : Set
    Edges : Vertex → Vertex → Set

open Graph_0

comap_0 : {A B : Set}
      → (f : A → B)
      → (Σ G : Graph_0 • Vertex G ≡ B)
      → (Σ H : Graph_0 • Vertex H ≡ A)
comap_0 {A} f (G , refl) = ⟨ A , (λ x y → Edges G (f x) (f y)) ⟩_0 , refl
```

Since the vertices are packed away as components of the records, the only way for f to refer to them is to awkwardly refer to seemingly arbitrary types, only then to have the vertices of the input graph G and the output graph H be constrained to match the type of the relabelling function f. Without the constraints, we could not even write the function for $\texttt{Graph}_0$. With such an importance, it is surprising to see that the occurrences of the constraint obligations are uninsightful refl-exivity proofs.

What the user would really want is to unbundle $\texttt{Graph}_0$ at will, to expose the first argument, to obtain $\texttt{Graph}_1$ below. Then, in stark contrast, the implementation $\texttt{comap}_1$ does not carry any excesses baggage at the type level nor at the implementation level.

```
record Graph₁ (Vertex : Set) : Set₁ where
  constructor ⟨_⟩₁
  field
    Edges : Vertex → Vertex → Set

comap₁ : {A B : Set}
       → (f : A → B)
       → Graph₁ B
       → Graph₁ A
comap₁ f ⟨ edges ⟩₁ = ⟨ (λ x y → edges (f x) (f y)) ⟩₁
```

With $\mathtt{Graph}_1$, one immediately sees that the comap operation "pulls back" the vertex type. Such an observation for $\mathtt{Graph}_0$ is not as easy; requiring familiarity with quantifier laws such as the one-point rule and quantifier distributivity.

A.15 Example: Graphs with Delayed Unbundling

The ubiquitous graph structure is contravariant in its collection of vertices. Recall that a multi-graph, or quiver, is a collection of vertices along with a collection of edges between any two vertices; here's the traditional record form:

```
Graph  : Context ℓ₁
Graph  = do Vertex ← Set
            Edges  ← (Vertex → Vertex → Set)
            End {ℓ₀}
```

Using the record form, it is awkward to phrase contravariance, which simply "relabels the vertices". Even worse, the awkward phrasing only serves to ensure certain constraints hold —which are reified at the value level via the uninsightful `refl`-exivity proof.

```
pattern ⟨_,_⟩ V E = (V , E , tt)

comap₀' : ∀ {A B : Set}
        → (f : A → B)
        → Σ G : Graph :kind ‘record • Field 0 G ≡ B
        → Σ G : Graph :kind ‘record • Field 0 G ≡ A
comap₀' {A} {B} f (⟨ .B , edgs ⟩ , refl) = (A , (λ a₁ a₂ → edgs (f a₁) (f a₂)) , tt) , refl
```

Without redefining graphs, we can phrase the definition at the 'typeclass' level —i.e., records parameterised by the vertices. This form is not only clearer and easier to implement at the value-level, it also makes it clear that we are "pulling back" the vertex type and so have also shown graphs are closed under reducts.

```
pattern ⟨_⟩¹ E = (E , tt)

-- Way better and less awkward!
comap' : ∀ {A B : Set}
       → (f : A → B)
       → (Graph :kind ‘typeclass) B
       → (Graph :kind ‘typeclass) A
comap' f ⟨ edgs ⟩¹ = ⟨ (λ a₁ a₂ → edgs (f a₁) (f a₂)) ⟩¹
```

Excellent, we can unbundle at will.