

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 $(M_1, _ \circ_1 _, Id_1)$ and $(M_2, _ \circ_2 _, Id_2)$, if we know¹ that $ceq : M_1 \equiv M_2$ then it is “obvious” that $Id_2 \circ_2 (x \circ_1 Id_1) \equiv x$ for all $x : M_1$. However, as written, this does not type-check. This is because $_ \circ_2 _$ 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 artificially prettier by defining `coe` to be `subst id ceq` without changing the heart of the matter. But if `Magma₀` 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.

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

```

50   record Magma1 (Carrier : Set) : Set where
51   field
52     _%_      : Carrier → Carrier → Carrier
53     Id       : Carrier
54
55   module Nicer
56     (M : Set)    {- The shared carrier -}
57     (A B : Magma1 M)
58   where
59     open Magma1 A renaming (Id to Id1; _%_ to _%1_ )
60     open Magma1 B renaming (Id to Id2; _%_ to _%2_ )
61
62     claim : ∀ x → Id2 %2 (x %1 Id1) ≡ x
63     claim = {!!}
64
65   This is the formaluation we expected, without noise. Thus it seems that it would be better to
66   expose the carrier. But, before long, we'd find a different concept, such as homomorphism, which
67   is awkward in this way, and cleaner using the first approach. These two approaches are called
68   bundled and unbundled respectively ?.
69
70   The definitions of homomorphism themselves (see below) is not so different, but the definition
71   of composition already starts to be quite unwieldly.
72
73   record Hom0 (A B : Magma0) : Set where ...
74   record Hom1 {M1 M2 : Set} (A : Magma1 M1) (B : Magma1 M2) : Set where ...
75
76   composition0 : ∀ {A B C} → Hom0 A B → Hom0 B C → Hom0 A C
77   composition0 = {!!}
78
79   composition1 : ∀ {M1 M2 M3} {A : Magma1 M1} {B : Magma1 M2} {C : Magma1 M3}
80     → Hom1 A B → Hom1 B C → Hom1 A C
81   composition1 = {!!}

```

This is the formaluation 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 ?.

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

```

73   record Hom0 (A B : Magma0) : Set where ...
74   record Hom1 {M1 M2 : Set} (A : Magma1 M1) (B : Magma1 M2) : Set where ...
75
76   composition0 : ∀ {A B C} → Hom0 A B → Hom0 B C → Hom0 A C
77   composition0 = {!!}
78
79   composition1 : ∀ {M1 M2 M3} {A : Magma1 M1} {B : Magma1 M2} {C : Magma1 M3}
80     → Hom1 A B → Hom1 B C → Hom1 A C
81   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., $\text{Monoid}_0 \cong \sum M : \text{Set} \bullet \text{Monoid}_1 M$. 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 application.

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 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, $_:\text{waist}_$, 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).

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 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
DynamicSystem_0	Set_1
DynamicSystem_1	$\Pi X : \text{Set} \bullet \text{Set}$
DynamicSystem_2	$\Pi X : \text{Set} \bullet \Pi x : X \bullet \text{Set}$

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* Set_1 to the *function type* $\Pi X : \text{Set} \bullet \text{Set}$ gets us from DynamicSystem_0 to something resembling DynamicSystem_1 , which we arrive at if we can obtain a *type constructor* $\lambda X : \text{Set} \bullet \dots$. We shall refer to the latter change as *reification* since the result is more concrete: It can be applied. This transformation will be denoted by $\Pi \rightarrow \lambda$. 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 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: $\text{id}_{i+1} = \Pi \rightarrow \lambda \text{id}_i$. The latter identity is proven by reflexivity in the appendices.

```

id0 : Set1
id0 =  $\Pi X : \mathbf{Set} \bullet \Pi e : X \bullet X$ 

id1 :  $\Pi X : \mathbf{Set} \bullet \mathbf{Set}$ 
id1 =  $\lambda (X : \mathbf{Set}) \rightarrow \Pi e : X \bullet X$ 

id2 :  $\Pi X : \mathbf{Set} \bullet \Pi e : X \bullet \mathbf{Set}$ 
id2 =  $\lambda (X : \mathbf{Set}) (e : X) \rightarrow X$ 

```

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  $\ell_1$ 
DynamicSystem = do State  $\leftarrow \mathbf{Set}$ 
                  start  $\leftarrow$  State
                  next  $\leftarrow$  (State  $\rightarrow$  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 $\mathbb{1}$ —which corresponds to \top in the Agda standard library and to $()$ in Haskell.

Exposure	Elaboration
0	$\Sigma \text{State} : \mathbf{Set} \bullet \Sigma \text{start} : X \bullet \Sigma \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$
1	$\Pi \text{State} : \mathbf{Set} \bullet \Sigma \text{start} : X \bullet \Sigma \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$
2	$\Pi \text{State} : \mathbf{Set} \bullet \Pi \text{start} : X \bullet \Sigma \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$
3	$\Pi \text{State} : \mathbf{Set} \bullet \Pi \text{start} : X \bullet \Pi \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$

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 =  $\lambda \ell \rightarrow \mathbb{N} \rightarrow \mathbf{Set} \ell$ 

{- Every type can be used as a context -}

```

```

197   ' _ : ∀ {ℓ} → Set ℓ → Context ℓ
198   ' S = λ _ → S

```

```

200   {- The “empty context” is the unit type -}
201   End : ∀ {ℓ} → Context ℓ
202   End = ' 1

```

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

```

207   _>>= : ∀ {a b}
208         → (Γ : Context a)
209         → (∀ {n} → Γ n → Context b)
210         → Context (a ⊔ b)
211   (Γ >>= f) zero    = Σ γ : Γ 0 • f γ 0
212   (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.

```

217   ' Set >>= λ State → ' State >>= λ start → ' (State → State) >>= λ next → End
218   {- or -}
219   do State ← ' Set
220     start ← ' State
221     next ← ' (State → State)
222   End

```

Interestingly [Bird 2009; Hudak et al. 2007], use of *do*-notation in preference to bind, $\gg=$, 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:

```

227   _>>= : ∀ {a b}
228         → (Γ : Set a) -- Main difference
229         → (Γ → Context b)
230         → Context (a ⊔ b)
231   (Γ >>= f) zero    = Σ γ : Γ • f γ 0
232   (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` $\not\cong$ `DynamicSystemi`, 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?

```

240   N0 : DynamicSystem 0 {- See the elaborations in Table 1 -}
241   N0 = λ , 0 , suc , tt
242
243   N1 : DynamicSystem 1
244   N1 = λ State → ??? {- Impossible to complete if “State” is empty! -}

```

```

246 {- "Instantiaing" X to be N in "DynamicSystem 1" -}
247 N1' : let State = N in Σ start : State • Σ s : (State → State) • 1
248 N1' = 0 , suc , tt

```

It seems what we need is a method, say $\Pi \rightarrow \lambda$, 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 $\Pi \rightarrow \lambda$. 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 $\Pi \rightarrow \lambda$ is thus a macro² that acts on the syntactic term representations of types. Below is main transformation —the details can be found in Appendix A.7.

$$\boxed{\Pi \rightarrow \lambda \ (\Pi \ a : A \bullet \tau) = (\lambda \ a : A \bullet \tau)}$$

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

```

250 Π → λ (Π → λ (DynamicSystem 2))
251 ≡ {- Definition of DynamicSystem at exposure level 2 -}
252 Π → λ (Π → λ (Π X : Set • Π s : X • Σ n : X → X • 1))
253 ≡ {- Definition of Π → λ -}
254 Π → λ (λ X : Set • Π s : X • Σ n : X → X • 1)
255 ≡ {- Homomorphism of Π → λ -}
256 λ X : Set • Π → λ (Π s : X • Σ n : X → X • 1)
257 ≡ {- Definition of Π → λ -}
258 λ X : Set • λ s : X • Σ n : X → X • 1

```

For practicality, `_:waist_` is a macro (defined in Appendix A.8) acting on contexts that repeats $\Pi \rightarrow \lambda$ a number of times in order to lift a number of field components to the parameter level.

```

259 τ :waist n = Π → λn (τ n)
260 f0 x = x
261 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 `< . . . >`.

```

262 -- Expressions of the form “... , tt” may now be written “< ... >”
263 infixr 5 < _>
264 < : ∀ {ℓ} → 1 {ℓ}
265 < = tt
266
267 < : ∀ {ℓ} {S : Set ℓ} → S → S
268 < s = s
269
270 <_ : ∀ {ℓ} {S : Set ℓ} → S → S × (1 {ℓ})
271 s > = s , tt

```

²A *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]

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

```

 $\mathcal{N}^0$  : DynamicSystem :waist 0
 $\mathcal{N}^0$  = ⟨  $\mathbb{N}$  , 0 , suc ⟩

 $\mathcal{N}^1$  : (DynamicSystem :waist 1)  $\mathbb{N}$ 
 $\mathcal{N}^1$  = ⟨ 0 , suc ⟩

 $\mathcal{N}^2$  : (DynamicSystem :waist 2)  $\mathbb{N}$  0
 $\mathcal{N}^2$  = ⟨ suc ⟩

 $\mathcal{N}^3$  : (DynamicSystem :waist 3)  $\mathbb{N}$  0 suc
 $\mathcal{N}^3$  = ⟨ ⟩

```

Using `:waist i` we may fix the first i -parameters ahead of time. Indeed, the type `(DynamicSystem :waist 1) \mathbb{N}` is the type of dynamic systems over carrier \mathbb{N} , whereas `(DynamicSystem :waist 2) \mathbb{N} 0` is the type of dynamic systems over carrier \mathbb{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 : ∀  $\ell$  → Context ( $\ell$  suc  $\ell$ )
Monoid  $\ell$  = do Carrier ← Set  $\ell$ 
             _ $\oplus$ _   ← (Carrier → Carrier → Carrier)
             Id     ← Carrier
             leftId  ← ∀ {x : Carrier} → x  $\oplus$  Id ≡ x
             rightId ← ∀ {x : Carrier} → Id  $\oplus$  x ≡ x
             assoc   ← ∀ {x y z} → (x  $\oplus$  y)  $\oplus$  z ≡ x  $\oplus$  (y  $\oplus$  z)
             End { $\ell$ }

```

With this context, `(Monoid ℓ_0 :waist 2) M \oplus` is the type of monoids over *particular* types M and *particular* operations \oplus . 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. for more on this idea. In particular, the description language \mathbb{D} for dynamical systems, below, declares concrete constructors for a fixpoint of a certain functor F ; i.e., $\mathbb{D} \cong \text{Fix } F$ where:

```

data  $\mathbb{D}$  : Set where
  startD :  $\mathbb{D}$ 
  nextD  :  $\mathbb{D}$  →  $\mathbb{D}$ 

F : Set → Set
F = λ (D : Set) → 1  $\uplus$  D

```

```

344 data Fix (F : Set → Set) : Set where
345   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 “ \Rightarrow ”):

```

348   do X ← Set; z ← X; s ← (X → X); End
349   ⇒ {- Use existing interpretation to obtain a record. -}
350     Σ X : Set • Σ z : X • Σ s : (X → X) • 1
351   ⇒ {- Pull out the carrier, “:waist 1”,
352        to obtain a type constructor using “Π→λ”. -}
353     λ X : Set • Σ z : X • Σ s : (X → X) • 1
354   ⇒ {- Termtypes constructors target the declared type,
355        so only their sources matter. E.g., ‘z : X’ is a
356        nullary constructor targeting the carrier ‘X’.
357        This introduces 1 types, so any existing
358        occurrences are dropped via 0. -}
359     λ X : Set • Σ z : 1 • Σ s : X • 0
360   ⇒ {- Termtypes are sums of products. -}
361     λ X : Set • 1 ⊔ X ⊔ 0
362   ⇒ {- Termtypes are fixpoints of type constructors. -}
363     Fix (λ X • 1 ⊔ X) -- 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 $\text{op} : A \rightarrow B$ targets the type termtype B and so its source is A . The details on the operations \Downarrow , $\Sigma \rightarrow \uplus$, and sources characterised by the pseudocode below can be found in appendices A.3.4, A.11.4, and A.11.3, respectively. It suffices to know that $\Sigma \rightarrow \uplus$ rewrites dependent-sums into sums, which requires the second argument to lose its reference to the first argument which is accomplished by \Downarrow ; further details can be found in the appendix.

```

374  ⌞⌋ τ = “reduce all de Bruijn indices within τ by 1”
375  ⌞⌋ τ = “reduce all de Bruijn indices within τ by 1”
376  Σ → ⊔ (Σ a : A • Ba) = A ⊔ Σ → ⊔ (⌞⌋ Ba)
377  sources (λ x : (Π a : A • Ba) • τ) = (λ x : A • sources τ)
378  sources (λ x : A • τ) = (λ x : 1 • sources τ)
379  sources (λ x : A • τ) = (λ x : 1 • sources τ)
380  termtype τ = Fix (Σ → ⊔ (sources τ))
381  ⌞⌋ τ = “reduce all de Bruijn indices within τ by 1”

```

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

```

385  D = termtype (DynamicSystem :waist 1)
386
387  -- Pattern synonyms for more compact presentation
388  pattern startD = μ (inj1 tt) -- : D
389  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-jections. For instance,

we can immediately see that the natural numbers act as the description language for dynamical systems:

```

to :  $\mathbb{D} \rightarrow \mathbb{N}$ 
to startD    = 0
to (nextD x) = suc (to x)

from :  $\mathbb{N} \rightarrow \mathbb{D}$ 
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 =  $\mu$  (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.11.6.

5 FREE DATATYPES FROM THEORIES

Astonishingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if $C : \mathbf{Set} \rightarrow \text{Context } \ell_0$ then $C' = \lambda X \rightarrow \text{termtype } (C \ X : \text{waist } 1)$ 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	\mathbb{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.

```

 $\mathbb{M} : \mathbf{Set}$ 
 $\mathbb{M} = \text{termtype } (\text{Monoid } \ell_0 : \text{waist } 1)$ 
{- ie Fix  $(\lambda X \rightarrow \mathbb{1}$ 
    -- Id, nil leaf
     $\uplus X \times X \times \mathbb{1}$  --  $\_ \oplus \_$ , branch
     $\uplus \mathbb{0}$            -- invariant leftId
     $\uplus \mathbb{0}$            -- invariant rightId
     $\uplus X \times X \times \mathbb{0}$  -- invariant assoc
     $\uplus \mathbb{0})$          -- the “End { $\ell$ }”
-}

-- Pattern synonyms for more compact presentation
pattern emptyM    =  $\mu$  (inj1 tt) -- :  $\mathbb{M}$ 
pattern branchM l r =  $\mu$  (inj2 (inj1 (l , r , tt))) -- :  $\mathbb{M} \rightarrow \mathbb{M} \rightarrow \mathbb{M}$ 
pattern absurdM a  =  $\mu$  (inj2 (inj2 (inj2 (inj2 a)))) -- absurd values of  $\mathbb{0}$ 

data TreeSkeleton :  $\mathbf{Set}$  where

```

```

442     empty  : TreeSkeleton
443     branch : TreeSkeleton → TreeSkeleton → TreeSkeleton

```

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

```

446     M→Tree : M → TreeSkeleton
447     M→Tree emptyM = empty
448     M→Tree (branchM l r) = branch (M→Tree l) (M→Tree r)
449     M→Tree (absurdM (inj1 ()))
450     M→Tree (absurdM (inj2 ()))
451
452     M←Tree : TreeSkeleton → M
453     M←Tree empty = emptyM
454     M←Tree (branch l r) = branchM (M←Tree l) (M←Tree r)
455

```

That these two operations are inverses is easily demonstrated.

```

456
457     M←Tree◦M→Tree : ∀ m → M←Tree (M→Tree m) ≡ m
458     M←Tree◦M→Tree emptyM = refl
459     M←Tree◦M→Tree (branchM l r) = cong2 branchM (M←Tree◦M→Tree l)
460                                     (M←Tree◦M→Tree r)
461
462     M←Tree◦M→Tree (absurdM (inj1 ()))
463     M←Tree◦M→Tree (absurdM (inj2 ()))
464
465     M→Tree◦M←Tree : ∀ t → M→Tree (M←Tree t) ≡ t
466     M→Tree◦M←Tree empty = refl
467     M→Tree◦M←Tree (branch l r) = cong2 branch (M→Tree◦M←Tree l)
468                                     (M→Tree◦M←Tree r)
469

```

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 trees over some ‘value type’ Ξ , one must start at the theory of “monoids containing a given set Ξ ”.

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.

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:

`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

	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

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

Table 4. Contexts as a semantics for PackageFormer constructs

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.12.

```
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	$\text{do } S \leftarrow \text{Set}; s \leftarrow S; n \leftarrow (S \rightarrow S); \text{End}$	“name-type pairs”
Record Type	$\sum S : \text{Set} \bullet \sum s : S \bullet \sum n : S \rightarrow S \bullet \mathbb{1}$	“bundled-up data”
Function Type	$\prod S \bullet \sum s : S \bullet \sum n : S \rightarrow S \bullet \mathbb{1}$	“a type of functions”
Type constructor	$\lambda S \bullet \sum s : S \bullet \sum n : S \rightarrow S \bullet \mathbb{1}$	“a function on types”
Algebraic datatype	$\text{data } \mathbb{D} : \text{Set} \text{ where } s : \mathbb{D}; n : \mathbb{D} \rightarrow \mathbb{D}$	“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 Tijis 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](https://arxiv.org/abs/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](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.

```
module Context where
```

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 ; [] ; _::_ ; _:_; sum)

ℓ₁ = Level.suc ℓ₀
```

A.2 Quantifiers Π and 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 \backslash : to obtain the “ghost colon” since standard colon $:$ is an Agda operator.

Even though Agda provides $\forall (x : \tau) \rightarrow fx$ 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 (_o_)

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

infix -666 Σ•
```

```

638 syntax  $\Sigma$  : • A ( $\lambda$  x  $\rightarrow$  B) =  $\Sigma$  x : A • B
639
640  $\Pi$  : •  $\forall$  {a b} (A : Set a) (B : A  $\rightarrow$  Set b)  $\rightarrow$  Set _
641  $\Pi$  : • A B = (x : A)  $\rightarrow$  B x
642
643 infix -666  $\Pi$  : •
644 syntax  $\Pi$  : • A ( $\lambda$  x  $\rightarrow$  B) =  $\Pi$  x : A • B
645
646 record  $\top$  { $\ell$ } : Set  $\ell$  where
647   constructor tt
648
649  $\perp$  =  $\top$  { $\ell_0$ }
650  $\emptyset$  =  $\perp$ 

```

A.3 Reflection

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

```

653 import Data.Unit as Unit
654 open import Reflection hiding (name; Type) renaming (_>=>_ to _>=>_m_)

```

A.3.1 Single argument application.

```

656 _app_ : Term  $\rightarrow$  Term  $\rightarrow$  Term
657 (def f args) app arg' = def f (args ::r arg (arg-info visible relevant) arg')
658 (con f args) app arg' = con f (args ::r arg (arg-info visible relevant) arg')
659 {-# CATCHALL #-}
660 tm app arg' = tm

```

Notice that we maintain existing applications:

$$\text{quoteTerm } (f \ x) \ \text{app} \ \text{quoteTerm } y \ \approx \ \text{quoteTerm } (f \ x \ y)$$

A.3.2 Reify \mathbb{N} term encodings as \mathbb{N} values.

```

665 toN : Term  $\rightarrow$   $\mathbb{N}$ 
666 toN (lit (nat n)) = n
667 {-# CATCHALL #-}
668 toN _ = 0

```

A.3.3 The Length of a Term.

```

671 arg-term :  $\forall$  { $\ell$ } {A : Set  $\ell$ }  $\rightarrow$  (Term  $\rightarrow$  A)  $\rightarrow$  Arg Term  $\rightarrow$  A
672 arg-term f (arg i x) = f x
673
674 {-# TERMINATING #-}
675 length $\ell$  : Term  $\rightarrow$   $\mathbb{N}$ 
676 length $\ell$  (var x args)      = 1 + sum (List.map (arg-term length $\ell$ ) args)
677 length $\ell$  (con c args)      = 1 + sum (List.map (arg-term length $\ell$ ) args)
678 length $\ell$  (def f args)      = 1 + sum (List.map (arg-term length $\ell$ ) args)
679 length $\ell$  (lam v (abs s x))  = 1 + length $\ell$  x
680 length $\ell$  (pat-lam cs args) = 1 + sum (List.map (arg-term length $\ell$ ) args)
681 length $\ell$  ( $\Pi$  [ x : A ] Bx) = 1 + length $\ell$  Bx
682 {-# CATCHALL #-}
683 -- sort, lit, meta, unknown
684 length $\ell$  t = 0

```

Here is an example use:

```

684 _ : length $\ell$  (quoteTerm ( $\Sigma$  x :  $\mathbb{N}$  • x  $\equiv$  x))  $\equiv$  10
685 _ = refl

```

A.3.4 Decreasing de Bruijn Indices. Given a quantification $(\oplus x : \tau \bullet fx)$, 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 .

```

687 var-dec0 : (fuel : ℕ) → Term → Term
688 var-dec0 zero t = t
689 -- Let's use an "impossible" term.
690 var-dec0 (suc n) (var zero args) = def (quote ⊥) []
691 var-dec0 (suc n) (var (suc x) args) = var x args
692 var-dec0 (suc n) (con c args) = con c (map-Args (var-dec0 n) args)
693 var-dec0 (suc n) (def f args) = def f (map-Args (var-dec0 n) args)
694 var-dec0 (suc n) (lam v (abs s x)) = lam v (abs s (var-dec0 n x))
695 var-dec0 (suc n) (pat-lam cs args) = pat-lam cs (map-Args (var-dec0 n) args)
696 var-dec0 (suc n) (Π[ s : arg i A ] B) = Π[ s : arg i (var-dec0 n A) ] var-dec0 n B
697 {-# CATCHALL #-}
698 -- sort, lit, meta, unknown
699 var-dec0 n t = t

```

In the paper proper, `var-dec` was mentioned once under the name \Downarrow .

```

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

```

Notice that we made the decision that x , the body of $(\oplus x \bullet x)$, will reduce to \emptyset , the empty type. Indeed, in such a situation the only Debruijn index cannot be reduced further. Here is an example:

```

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

```

A.4 Context Monad

```

710 Context = λ ℓ → ℕ → Set ℓ
711
712 infix -1000 ' _
713 ' _ : ∀ {ℓ} → Set ℓ → Context ℓ
714 ' S = λ _ → S
715
716 End : ∀ {ℓ} → Context ℓ
717 End = ' ⊤
718
719 End0 = End {ℓ0}
720
721 _>=>_ : ∀ {a b}
722       → (Γ : Set a) -- Main difference
723       → (Γ → Context b)
724       → Context (a ⊔ b)
725 (Γ >=> f) ℕ.zero = Σ γ : Γ • f γ ⊔
726 (Γ >=> f) (suc n) = (γ : Γ) → f γ n

```

A.5 ⟨⟩ Notation

As mentioned, grouping mechanisms are declared with `do . . . End`, and instances of them are constructed using `⟨ . . . ⟩`.

```

729 -- Expressions of the form "... , tt" may now be written "{ ... }"
730 infixr 5 ⟨ _⟩
731 ⟨ _ : ∀ {ℓ} → T {ℓ}
732 ⟨ = tt
733
734 ⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
735 ⟨ s = s

```

```

736  _> : ∀ {ℓ} {S : Set ℓ} → S → S × T {ℓ}
737  s ) = s , tt
738

```

A.6 DynamicSystem Context

```

739  DynamicSystem : Context (ℓsuc Level.zero)
740  DynamicSystem = do X ← Set
741                  z ← X
742                  s ← (X → X)
743                  End {Level.zero}
744
745  -- Records with n-Parameters, n : 0..3
746  A B C D : Set1
747  A = DynamicSystem 0 -- Σ X : Set • Σ z : X • Σ s : X → X • T
748  B = DynamicSystem 1 -- (X : Set) → Σ z : X • Σ s : X → X • T
749  C = DynamicSystem 2 -- (X : Set) (z : X) → Σ s : X → X • T
750  D = DynamicSystem 3 -- (X : Set) (z : X) → (s : X → X) → T
751
752  _ : A ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • T) ; _ = refl
753  _ : B ≡ (Π X : Set • Σ z : X • Σ s : (X → X) • T) ; _ = refl
754  _ : C ≡ (Π X : Set • Π z : X • Σ s : (X → X) • T) ; _ = refl
755  _ : D ≡ (Π X : Set • Π z : X • Π s : (X → X) • T) ; _ = refl
756
757  stability : ∀ {n} → DynamicSystem (3 + n)
758             ≡ DynamicSystem 3
759  stability = refl
760
761  B-is-empty : ¬ B
762  B-is-empty b = proj1( b ⊥ )
763
764  N0 : DynamicSystem 0
765  N0 = ℕ , 0 , suc , tt
766
767  N : DynamicSystem 0
768  N = ⟨ ℕ , 0 , suc ⟩
769
770  B-on-ℕ : Set
771  B-on-ℕ = let X = ℕ in Σ z : X • Σ s : (X → X) • T
772
773  ex : B-on-ℕ
774  ex = ⟨ 0 , suc ⟩
775

```

A.7 Π→λ

```

776  Π→λ-helper : Term → Term
777  Π→λ-helper (pi a b) = lam visible b
778  Π→λ-helper (lam a (abs x y)) = lam a (abs x (Π→λ-helper y))
779  {-# CATCHALL #-}
780  Π→λ-helper x = x
781
782  macro
783  Π→λ : Term → Term → TC Unit.T
784  Π→λ tm goal = normalise tm >=>m λ tm' → unify (Π→λ-helper tm') goal

```

A.8 _:waist_

```

785  waist-helper : ℕ → Term → Term
786  waist-helper zero t = t

```



```

785     waist-helper (suc n) t = waist-helper n (II→λ-helper t)
786
787   macro
788     λ:waist_ : Term → Term → Term → TC Unit.⊤
789     λ:waist_ t n goal =      normalise (t app n)
790                               >>=m λ t' → unify (waist-helper (toN n) t') goal

```

A.9 DynamicSystem :waist i

```

792   A' : Set1
793   B' : ∀ (X : Set) → Set
794   C' : ∀ (X : Set) (x : X) → Set
795   D' : ∀ (X : Set) (x : X) (s : X → X) → Set
796
797   A' = DynamicSystem :waist 0
798   B' = DynamicSystem :waist 1
799   C' = DynamicSystem :waist 2
800   D' = DynamicSystem :waist 3
801
802   N0 : A'
803   N0 = ⟨ N , 0 , suc ⟩
804
805   N1 : B' N
806   N1 = ⟨ 0 , suc ⟩
807
808   N2 : C' N 0
809   N2 = ⟨ suc ⟩
810
811   N3 : D' N 0 suc
812   N3 = ⟨ ⟩

```

It may be the case that $\Gamma \ 0 \equiv \Gamma \text{ :waist } 0$ for every context Γ .

```

811   _ : DynamicSystem 0 ≡ DynamicSystem :waist 0
812   _ = refl

```

A.10 Field projections

```

815   Field0 : N → Term → Term
816   Field0 zero c    = def (quote proj1) (arg (arg-info visible relevant) c :: [])
817   Field0 (suc n) c = Field0 n (def (quote proj2) (arg (arg-info visible relevant) c :: []))
818
819   macro
820     Field : N → Term → Term → TC Unit.⊤
821     Field n t goal = unify goal (Field0 n t)

```

A.11 Termtypes

Using the guide, ??, outlined in the paper proper we shall form D_i for each stage in the calculation.

A.11.1 Stage 1: Records.

```

826   D1 = DynamicSystem 0
827
828   1-records : D1 ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • ⊤)
829   1-records = refl

```

A.11.2 Stage 2: Parameterised Records.

```

831   D2 = DynamicSystem :waist 1

```

```

834 2-funcs : D2 ≡ (λ (X : Set) → Σ z : X • Σ s : (X → X) • T)
835 2-funcs = refl

```

A.11.3 *Stage 3: Sources.* Let's begin with an example to motivate the definition of sources.

```

837 _ : quoteTerm (V {x : N} → N)
838   ≡ pi (arg (arg-info hidden relevant) (quoteTerm N)) (abs "x" (quoteTerm N))
839 _ = refl

```

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

```

843 sources0 : Term → Term
844 -- Otherwise:
845 sources0 (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) =
846   def (quote _X_) (vArg A
847     :: vArg (def (quote _X_)
848       (vArg (var-dec Ba) :: vArg (var-dec (var-dec (sources0 Cab))) :: []))
849     :: [])
850 sources0 (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 0
851 sources0 (Π[ x : arg i A ] Bx) = A
852 {-# CATCHALL #-}
853 -- sort, lit, meta, unknown
854 sources0 t = quoteTerm 1
855
856 {-# TERMINATING #-}
857 sources1 : Term → Term
858 sources1 (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 0
859 sources1 (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) = def (quote _X_) (vArg A ::
860   vArg (def (quote _X_) (vArg (var-dec Ba) :: vArg (var-dec (var-dec (sources0 Cab))) :: [])) :: [])
861 sources1 (Π[ x : arg i A ] Bx) = A
862 sources1 (def (quote Σ) (ℓ1 :: ℓ2 :: τ :: body))
863   = def (quote Σ) (ℓ1 :: ℓ2 :: map-Arg sources0 τ :: List.map (map-Arg sources1) body)
864 -- This function introduces 1s, so let's drop any old occurrences a la 0.
865 sources1 (def (quote T) _) = def (quote 0) []
866 sources1 (lam v (abs s x)) = lam v (abs s (sources1 x))
867 sources1 (var x args) = var x (List.map (map-Arg sources1) args)
868 sources1 (con c args) = con c (List.map (map-Arg sources1) args)
869 sources1 (def f args) = def f (List.map (map-Arg sources1) args)
870 sources1 (pat-lam cs args) = pat-lam cs (List.map (map-Arg sources1) args)
871 {-# CATCHALL #-}
872 -- sort, lit, meta, unknown
873 sources1 t = t

```

We now form the macro and some unit tests.

```

874 macro
875   sources : Term → Term → TC Unit.T
876   sources tm goal = normalise tm >>= m λ tm' → unify (sources1 tm') goal
877
878 _ : sources (N → Set) ≡ N
879 _ = refl
880
881 _ : sources (Σ x : (N → Fin 3) • N) ≡ (Σ x : N • N)
882 _ = refl
883
884 _ : V {ℓ : Level} {A B C : Set}
885   → sources (Σ x : (A → B) • C) ≡ (Σ x : A • C)
886 _ = refl

```

```

883   _ : sources (Fin 1 → Fin 2 → Fin 3) ≡ (Σ _ : Fin 1 • Fin 2 × 1)
884   _ = refl
885
886   _ : sources (Σ f : (Fin 1 → Fin 2 → Fin 3 → Fin 4) • Fin 5)
887   ≡ (Σ f : (Fin 1 × Fin 2 × Fin 3) • Fin 5)
888   _ = refl
889
890   _ : ∀ {A B C : Set} → sources (A → B → C) ≡ (A × B × 1)
891   _ = refl
892
893   _ : ∀ {A B C D E : Set} → sources (A → B → C → D → E)
894   ≡ Σ A (λ _ → Σ B (λ _ → Σ C (λ _ → Σ D (λ _ → T))))
895   _ = refl

```

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

```

896   -- one implicit
897   _ : sources (∀ {x : N} → x ≡ x) ≡ 0
898   _ = refl
899
900   -- multiple implicits
901   _ : sources (∀ {x y z : N} → x ≡ y) ≡ 0
902   _ = refl

```

The third stage can now be formed.

```

903   D3 = sources D2
904
905   3-sources : D3 ≡ λ (X : Set) → Σ z : 1 • Σ s : X • 0
906   3-sources = refl

```

A.11.4 Stage 4: $\Sigma \rightarrow \cup$ -Replacing Products with Sums.

```

907   {-# TERMINATING #-}
908   Σ→0 : Term → Term
909   Σ→0 (def (quote Σ) (h1 :: h0 :: arg i A :: arg i1 (lam v (abs s x)) :: []))
910   = def (quote _0) (h1 :: h0 :: arg i A :: vArg (Σ→0 (var-dec x)) :: [])
911   -- Interpret "End" in do-notation to be an empty, impossible, constructor.
912   Σ→0 (def (quote T) _) = def (quote ⊥) []
913   -- Walk under λ's and Π's.
914   Σ→0 (lam v (abs s x)) = lam v (abs s (Σ→0 x))
915   Σ→0 (Π[ x : A ] Bx) = Π[ x : A ] Σ→0 Bx
916   {-# CATCHALL #-}
917   Σ→0 t = t
918
919   macro
920     Σ→0 : Term → Term → TC Unit.T
921     Σ→0 tm goal = normalise tm >>= m λ tm' → unify (Σ→0 tm') goal
922
923   -- Unit tests
924   _ : Σ→0 (Π X : Set • (X → X)) ≡ (Π X : Set • (X → X)); _ = refl
925   _ : Σ→0 (Π X : Set • Σ s : X • X) ≡ (Π X : Set • X ⊔ X) ; _ = refl
926   _ : Σ→0 (Π X : Set • Σ s : (X → X) • X) ≡ (Π X : Set • (X → X) ⊔ X) ; _ = refl
927   _ : Σ→0 (Π X : Set • Σ z : X • Σ s : (X → X) • T {ℓ0}) ≡ (Π X : Set • X ⊔ (X → X) ⊔ ⊥) ; _ = refl
928
929   D4 = Σ→0 D3
930
931   4-unions : D4 ≡ λ X → 1 ⊔ X ⊔ 0
932   4-unions = refl

```

A.11.5 Stage 5: Fixpoint and proof that $\mathbb{D} \cong \mathbb{N}$.

```

932 {-# NO_POSITIVITY_CHECK #-}
933 data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
934   μ : F (Fix F) → Fix F
935
936 ℙ = Fix D4
937
938 -- Pattern synonyms for more compact presentation
939 pattern zeroD = μ (inj1 tt) -- : ℙ
940 pattern sucD e = μ (inj2 (inj1 e)) -- : ℙ → ℙ
941
942 to : ℙ → ℕ
943 to zeroD = 0
944 to (sucD x) = suc (to x)
945
946 from : ℕ → ℙ
947 from zero = zeroD
948 from (suc n) = sucD (from n)
949
950 toofrom : ∀ n → to (from n) ≡ n
951 toofrom zero = refl
952 toofrom (suc n) = cong suc (toofrom n)
953
954 fromoto : ∀ d → from (to d) ≡ d
955 fromoto zeroD = refl
956 fromoto (sucD x) = cong sucD (fromoto x)

```

A.11.6 *termtype and Inj macros*. We summarise the stages together into one macro: “termtype : UnaryFunctor → Type”.

```

956 macro
957   termtype : Term → Term → TC Unit.T
958   termtype tm goal =
959     normalise tm
960     >>= 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”.

```

964 Inj0 : ℕ → Term → Term
965 Inj0 zero c = con (quote inj1) (arg (arg-info visible relevant) c :: [])
966 Inj0 (suc n) c = con (quote inj2) (vArg (Inj0 n c) :: [])
967
968 -- Duality!
969 -- i-th projection: proj1 ∘ (proj2 ∘ ⋯ ∘ proj2)
970 -- i-th injection: (inj2 ∘ ⋯ ∘ inj2) ∘ inj1
971
972 macro
973   Inj : ℕ → Term → Term → TC Unit.T
974   Inj n t goal = unify goal ((con (quote μ) []) app (Inj0 n t))

```

With this alternative, we regain the “user chosen constructor names” for ℙ:

```

975 startD : ℙ
976 startD = Inj 0 (tt {ℓ0})
977
978 nextD' : ℙ → ℙ
979 nextD' d = Inj 1 d

```

A.12 `:kind`

```

981  data Kind : Set where
982      'record   : Kind
983      'typeclass : Kind
984      'data     : Kind
985
986  macro
987      _:kind_ : Term → Term → Term → TC Unit.⊤
988      _:kind_ t (con (quote 'record) _) goal = normalise (t app (quoteTerm 0))
989              >=>_m λ t' → unify (waist-helper 0 t') goal
990      _:kind_ t (con (quote 'typeclass) _) goal = normalise (t app (quoteTerm 1))
991              >=>_m λ t' → unify (waist-helper 1 t') goal
992      _:kind_ t (con (quote 'data) _) goal = normalise (t app (quoteTerm 1))
993              >=>_m λ t' → normalise (waist-helper 1 t')
994              >=>_m λ t'' → unify goal (def (quote Fix) ((vArg (Σ→ℳ₀ (sources₁ t'')))) :: []))
995      _:kind_ t _ goal = unify t goal

```

Informally, `_:kind_` behaves as follows:

```

996  C :kind 'record   = C :waist 0
997  C :kind 'typeclass = C :waist 1
998  C :kind 'data     = termtype (C :waist 1)

```

A.13 `termtype PointedSet ≅ 1`

```

1000  -- termtype (PointedSet) ≅ 1 !
1001  One : Context (ℓsuc ℓ₀)
1002  One   = do Carrier ← Set ℓ₀
1003          point ← Carrier
1004          End {ℓ₀}
1005
1006  One : Set
1007  One = termtype (One :waist 1)
1008
1009  view₁ : One → 1
1010  view₁ emptyM = tt

```