

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
66

```

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.

```

70 record Hom0 (A B : Magma0) : Set where ...
71 record Hom1 {M1 M2 : Set} (A : Magma1 M1) (B : Magma1 M2) : Set where ...
72
73 composition0 : ∀ {A B C} → Hom0 A B → Hom0 B C → Hom0 A C
74 composition0 = {!!}
75
76 composition1 : ∀ {M1 M2 M3} {A : Magma1 M1} {B : Magma1 M2} {C : Magma1 M3}
77   → Hom1 A B → Hom1 B C → Hom1 A C
78 composition1 = {!!}
79
80
81

```

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 =  $\Pi X : \text{Set} \bullet \Pi e : X \bullet X$ 

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

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

```

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  $\leftarrow \text{Set}$ 
                  start  $\leftarrow \text{State}$ 
                  next  $\leftarrow (\text{State} \rightarrow \text{State})$ 
                  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	$\Sigma \text{State} : \text{Set} \bullet \Sigma \text{start} : X \bullet \Sigma \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$
1	$\Pi \text{State} : \text{Set} \bullet \Sigma \text{start} : X \bullet \Sigma \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$
2	$\Pi \text{State} : \text{Set} \bullet \Pi \text{start} : X \bullet \Sigma \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$
3	$\Pi \text{State} : \text{Set} \bullet \Pi \text{start} : X \bullet \Pi \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$

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

{- 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  ⎵
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  termtype τ = Fix (Σ → ⊔ (sources τ))

```

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:

```

393 to :  $\mathbb{D} \rightarrow \mathbb{N}$ 
394 to startD = 0
395 to (nextD x) = suc (to x)
396
397
398
399 from :  $\mathbb{N} \rightarrow \mathbb{D}$ 
400 from zero = startD
401 from (suc n) = nextD (from n)
402

```

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 a parameterised context $C : \mathbf{Set} \rightarrow \text{Context } \ell_0$ is given, then

```

412  $\mathbb{C} = \lambda X \rightarrow \text{termtype } (C X : \text{waist } 1)$ 
413

```

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.

```

426  $\mathbb{M} : \mathbf{Set}$ 
427  $\mathbb{M} = \text{termtype } (\text{Monoid } \ell_0 : \text{waist } 1)$ 
428 {- i.e., Fix ( $\lambda X \rightarrow \mathbb{1}$  -- Id, nil leaf
429            $\uplus X \times X \times \mathbb{1}$  --  $\_ \oplus \_$ , branch
430            $\uplus \mathbb{0}$  -- invariant leftId
431            $\uplus \mathbb{0}$  -- invariant rightId
432            $\uplus X \times X \times \mathbb{0}$  -- invariant assoc
433            $\uplus \mathbb{0}$ ) -- the “End { $\ell$ }”
434 -}
435
436 -- Pattern synonyms for more compact presentation
437 pattern emptyM =  $\mu$  (inj1 tt) -- :  $\mathbb{M}$ 
438 pattern branchM l r =  $\mu$  (inj2 (inj1 (l , r , tt))) -- :  $\mathbb{M} \rightarrow \mathbb{M} \rightarrow \mathbb{M}$ 
439 pattern absurdM a =  $\mu$  (inj2 (inj2 (inj2 (inj2 a)))) -- absurd values of  $\mathbb{0}$ 
440
441

```

```

442 data TreeSkeleton : Set where
443   empty : TreeSkeleton
444   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`.

```

447 M→Tree : M → TreeSkeleton
448 M→Tree emptyM = empty
449 M→Tree (branchM l r) = branch (M→Tree l) (M→Tree r)
450 M→Tree (absurdM (inj1 ()))
451 M→Tree (absurdM (inj2 ()))
452
453 M←Tree : TreeSkeleton → M
454 M←Tree empty = emptyM
455 M←Tree (branch l r) = branchM (M←Tree l) (M←Tree r)
456

```

That these two operations are inverses is easily demonstrated.

```

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

```

475 Collection : ∀ ℓ → Context (ℓsuc ℓ)
476 Collection ℓ = do Elem ← Set ℓ
477                   Carrier ← Set ℓ
478                   insert ← (Elem → Carrier → Carrier)
479                   ∅ ← Carrier
480                   End {ℓ}
481
482 C : Set → Set
483 C Elem = termtree ((Collection ℓ0 :waist 2) Elem)
484
485 pattern _::_ x xs = μ (inj1 (x , xs , tt))
486 pattern ∅ = μ (inj2 (inj1 tt))
487
488 to : ∀ {E} → C E → List E
489 to (e :: es) = e :: to es
490

```

```

491   to  $\emptyset$  = []
492
493   from :  $\forall \{E\} \rightarrow \text{List } E \rightarrow \mathbb{C} \ E$ 
494   from [] =  $\emptyset$ 
495   from (x :: xs) = x :: from xs
496
497   toofrom :  $\forall \{E\} \ (xs : \text{List } E) \rightarrow \text{to } (\text{from } xs) \equiv xs$ 
498   toofrom [] = refl
499   toofrom (x :: xs) = cong (x ::_) (toofrom xs)
500
501   fromoto :  $\forall \{E\} \ (e : \mathbb{C} \ E) \rightarrow \text{from } (\text{to } e) \equiv e$ 
502   fromoto (e :: es) = cong (e ::_) (fromoto es)
503   fromoto  $\emptyset$  = refl
504

```

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:

`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.12.

```

537   data Kind : Set where
538     'record   : Kind
539

```

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

Table 4. Contexts as a semantics for PackageFormer constructs

```

'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 mechanism 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	$\Sigma S : \text{Set} \bullet \Sigma s : S \bullet \Sigma n : S \rightarrow S \bullet \mathbb{1}$	“bundled-up data”
Function Type	$\Pi S \bullet \Sigma s : S \bullet \Sigma n : S \rightarrow S \bullet \mathbb{1}$	“a type of functions”
Type constructor	$\lambda S \bullet \Sigma s : S \bullet \Sigma n : S \rightarrow S \bullet \mathbb{1}$	“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 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](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 ; [] ; _::_ ; _::^r_; sum)

ℓ₁ = Level.suc ℓ₀
```

A.2 Quantifiers $\Pi\bullet/\Sigma\bullet$ 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•
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 T {ℓ} : Set ℓ where
  constructor tt

⊥ = T {ℓ₀}
⊥ = ⊥
```

687 A.3 Reflection

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

```
689 import Data.Unit as Unit
690 open import Reflection hiding (name; Type) renaming (<_>=_ to <_>=_m_)
```

691 A.3.1 Single argument application.

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

697 Notice that we maintain existing applications:

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

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

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

705 A.3.3 The Length of a Term.

```
706 arg-term : ∀ {ℓ} {A : Set ℓ} → (Term → A) → Arg Term → A
707 arg-term f (arg i x) = f x
708
709 {-# TERMINATING #-}
710 lengtht : Term → ℕ
711 lengtht (var x args)      = 1 + sum (List.map (arg-term lengtht) args)
712 lengtht (con c args)      = 1 + sum (List.map (arg-term lengtht) args)
713 lengtht (def f args)      = 1 + sum (List.map (arg-term lengtht) args)
714 lengtht (lam v (abs s x)) = 1 + lengtht x
715 lengtht (pat-lam cs args) = 1 + sum (List.map (arg-term lengtht) args)
716 lengtht (Π[ x : A ] Bx)    = 1 + lengtht Bx
717 {-# CATCHALL #-}
718 -- sort, lit, meta, unknown
719 lengtht t = 0
```

719 Here is an example use:

```
720 _ : lengtht (quoteTerm (Σ x : ℕ • x ≡ x)) ≡ 10
721 _ = refl
```

722 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 .

```
723 var-dec0 : (fuel : ℕ) → Term → Term
724 var-dec0 zero t = t
725 -- Let's use an "impossible" term.
726 var-dec0 (suc n) (var zero args) = def (quote ⊥) []
727 var-dec0 (suc n) (var (suc x) args) = var x args
728 var-dec0 (suc n) (con c args) = con c (map-Args (var-dec0 n) args)
729 var-dec0 (suc n) (def f args) = def f (map-Args (var-dec0 n) args)
730 var-dec0 (suc n) (lam v (abs s x)) = lam v (abs s (var-dec0 n x))
731 var-dec0 (suc n) (pat-lam cs args) = pat-lam cs (map-Args (var-dec0 n) args)
732 var-dec0 (suc n) (Π[ s : arg i A ] B) = Π[ s : arg i (var-dec0 n A) ] var-dec0 n B
733 {-# CATCHALL #-}
```

```

736 -- sort, lit, meta, unknown
737 var-dec0 n t = t

```

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

```

739 var-dec : Term → Term
740 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 Debrujin index cannot be reduced further. Here is an example:

```

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

```

A.4 Context Monad

```

746 Context = λ ℓ → N → Set ℓ

748 infix -1000 ' _
749 ' _ : ∀ {ℓ} → Set ℓ → Context ℓ
750 ' S = λ _ → S

752 End : ∀ {ℓ} → Context ℓ
753 End = ' ⊤

754 End0 = End {ℓ0}

756 _>=>_ : ∀ {a b}
757   → (Γ : Set a) -- Main difference
758   → (Γ → Context b)
759   → Context (a ∪ b)
760 (Γ >=> f) N.zero = Σ γ : Γ • f γ ∅
761 (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 `⟨ . . . ⟩`.

```

765 -- Expressions of the form "... , tt" may now be written "< ... >"
766 infixr 5 < _>
767 < _ : ∀ {ℓ} → T {ℓ}
768 < _ = tt

769 < _ : ∀ {ℓ} {S : Set ℓ} → S → S
770 < s = s

772 _> : ∀ {ℓ} {S : Set ℓ} → S → S × T {ℓ}
773 s > = s , tt

```

A.6 DynamicSystem Context

```

775 DynamicSystem : Context (ℓsuc Level.zero)
776 DynamicSystem = do X ← Set
777                 z ← X
778                 s ← (X → X)
779                 End {Level.zero}

780
781 -- Records with n-Parameters, n : 0..3
782 A B C D : Set1
783 A = DynamicSystem 0 -- Σ X : Set • Σ z : X • Σ s : X → X • T
784 B = DynamicSystem 1 -- (X : Set) → Σ z : X • Σ s : X → X • T

```



```

785 C = DynamicSystem 2 -- (X : Set) (z : X) →  $\Sigma$  s : X → X • T
786 D = DynamicSystem 3 -- (X : Set) (z : X) → (s : X → X) → T
787
788 _ : A ≡ ( $\Sigma$  X : Set •  $\Sigma$  z : X •  $\Sigma$  s : (X → X) • T) ; _ = refl
789 _ : B ≡ ( $\Pi$  X : Set •  $\Sigma$  z : X •  $\Sigma$  s : (X → X) • T) ; _ = refl
790 _ : C ≡ ( $\Pi$  X : Set •  $\Pi$  z : X •  $\Sigma$  s : (X → X) • T) ; _ = refl
791 _ : D ≡ ( $\Pi$  X : Set •  $\Pi$  z : X •  $\Pi$  s : (X → X) • T) ; _ = refl
792
793 stability :  $\forall \{n\} \rightarrow$  DynamicSystem (3 + n)
794           ≡ DynamicSystem 3
795
796 B-is-empty :  $\neg$  B
797 B-is-empty b = proj1( b  $\perp$  )
798
799  $\mathcal{N}_0$  : DynamicSystem 0
800  $\mathcal{N}_0$  =  $\mathbb{N}$  ,  $\emptyset$  , suc , tt
801
802  $\mathcal{N}$  : DynamicSystem 0
803  $\mathcal{N}$  =  $\langle \mathbb{N} , \emptyset , \text{suc} \rangle$ 
804
805 B-on- $\mathbb{N}$  : Set
806 B-on- $\mathbb{N}$  = let X =  $\mathbb{N}$  in  $\Sigma$  z : X •  $\Sigma$  s : (X → X) • T
807
808 ex : B-on- $\mathbb{N}$ 
809 ex =  $\langle \emptyset , \text{suc} \rangle$ 

```

A.7 $\Pi \rightarrow \lambda$

```

809  $\Pi \rightarrow \lambda$ -helper : Term → Term
810  $\Pi \rightarrow \lambda$ -helper (pi a b) = lam visible b
811  $\Pi \rightarrow \lambda$ -helper (lam a (abs x y)) = lam a (abs x ( $\Pi \rightarrow \lambda$ -helper y))
812 {-# CATCHALL #-}
813  $\Pi \rightarrow \lambda$ -helper x = x
814
815 macro
816  $\Pi \rightarrow \lambda$  : Term → Term → TC Unit.T
817  $\Pi \rightarrow \lambda$  tm goal = normalise tm >=>_m  $\lambda$  tm' → unify ( $\Pi \rightarrow \lambda$ -helper tm') goal

```

A.8 $_:\text{waist}__$

```

818 waist-helper :  $\mathbb{N} \rightarrow$  Term → Term
819 waist-helper zero t = t
820 waist-helper (suc n) t = waist-helper n ( $\Pi \rightarrow \lambda$ -helper t)
821
822 macro
823  $\_:\text{waist}_\_$  : Term → Term → Term → TC Unit.T
824  $\_:\text{waist}_\_$  t n goal = normalise (t app n)
825                       >=>_m  $\lambda$  t' → unify (waist-helper (to $\mathbb{N}$  n) t') goal

```

A.9 DynamicSystem :waist i

```

827 A' : Set1
828 B' :  $\forall$  (X : Set) → Set
829 C' :  $\forall$  (X : Set) (x : X) → Set
830 D' :  $\forall$  (X : Set) (x : X) (s : X → X) → Set
831
832 A' = DynamicSystem :waist 0
833 B' = DynamicSystem :waist 1

```

```

834 C' = DynamicSystem :waist 2
835 D' = DynamicSystem :waist 3
836
837  $\mathcal{N}^0$  : A'
838  $\mathcal{N}^0$  = ⟨  $\mathbb{N}$  , 0 , suc ⟩
839
840  $\mathcal{N}^1$  : B'  $\mathbb{N}$ 
841  $\mathcal{N}^1$  = ⟨ 0 , suc ⟩
842
843  $\mathcal{N}^2$  : C'  $\mathbb{N}$  0
844  $\mathcal{N}^2$  = ⟨ suc ⟩
845
846  $\mathcal{N}^3$  : D'  $\mathbb{N}$  0 suc
847  $\mathcal{N}^3$  = ⟨ ⟩

```

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

```

847 _ : DynamicSystem 0 ≡ DynamicSystem :waist 0
848 _ = refl

```

850 A.10 Field projections

```

851 Field0 :  $\mathbb{N} \rightarrow \text{Term} \rightarrow \text{Term}$ 
852 Field0 zero c = def (quote proj1) (arg (arg-info visible relevant) c :: [])
853 Field0 (suc n) c = Field0 n (def (quote proj2) (arg (arg-info visible relevant) c :: []))
854
855 macro
856   Field :  $\mathbb{N} \rightarrow \text{Term} \rightarrow \text{Term} \rightarrow \text{TC Unit.T}$ 
857   Field n t goal = unify goal (Field0 n t)

```

858 A.11 Termtypes

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

860 A.11.1 Stage 1: Records.

```

861 D1 = DynamicSystem 0
862
863 1-records : D1 ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • T)
864 1-records = refl

```

866 A.11.2 Stage 2: Parameterised Records.

```

867 D2 = DynamicSystem :waist 1
868
869 2-funcs : D2 ≡ (λ (X : Set) → Σ z : X • Σ s : (X → X) • T)
870 2-funcs = refl

```

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

```

872 _ : quoteTerm (V {x :  $\mathbb{N}$ } →  $\mathbb{N}$ )
873 ≡ pi (arg (arg-info hidden relevant) (quoteTerm  $\mathbb{N}$ )) (abs "x" (quoteTerm  $\mathbb{N}$ ))
874 _ = refl

```

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

```

877 sources0 : Term → Term
878 -- Otherwise:
879 sources0 (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) =
880   def (quote _X_) (vArg A
881     :: vArg (def (quote _X_)
882       (vArg (var-dec Ba) :: vArg (var-dec (var-dec (sources0 Cab))) :: []))

```

```

883             :: [])
884 sources0 (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 0
885 sources0 (Π[ x : arg i A ] Bx) = A
886 {-# CATCHALL #-}
887 -- sort, lit, meta, unknown
888 sources0 t = quoteTerm 1
889
890 {-# TERMINATING #-}
891 sources1 : Term → Term
892 sources1 (Π[ a : arg (arg-info hidden _) A ] Ba) = quoteTerm 0
893 sources1 (Π[ a : arg i A ] (Π[ b : arg _ Ba ] Cab)) = def (quote _X_) (vArg A ::
894   vArg (def (quote _X_) (vArg (var-dec Ba) :: vArg (var-dec (sources0 Cab))) :: [])) :: [])
895 sources1 (Π[ x : arg i A ] Bx) = A
896 sources1 (def (quote Σ) (ℓ1 :: ℓ2 :: τ :: body))
897   = def (quote Σ) (ℓ1 :: ℓ2 :: map-Arg sources0 τ :: List.map (map-Arg sources1) body)
898 -- This function introduces 1s, so let's drop any old occurrences a la 0.
899 sources1 (def (quote T) _) = def (quote 0) []
900 sources1 (lam v (abs s x)) = lam v (abs s (sources1 x))
901 sources1 (var x args) = var x (List.map (map-Arg sources1) args)
902 sources1 (con c args) = con c (List.map (map-Arg sources1) args)
903 sources1 (def f args) = def f (List.map (map-Arg sources1) args)
904 sources1 (pat-lam cs args) = pat-lam cs (List.map (map-Arg sources1) args)
905 {-# CATCHALL #-}
906 -- sort, lit, meta, unknown
907 sources1 t = t

```

We now form the macro and some unit tests.

```

905 macro
906   sources : Term → Term → TC Unit.T
907   sources tm goal = normalise tm >>= m λ tm' → unify (sources1 tm') goal
908
909 _ : sources (N → Set) ≡ N
910 _ = refl
911
912 _ : sources (Σ x : (N → Fin 3) • N) ≡ (Σ x : N • N)
913 _ = refl
914
915 _ : ∀ {ℓ : Level} {A B C : Set}
916   → sources (Σ x : (A → B) • C) ≡ (Σ x : A • C)
917 _ = refl
918
919 _ : sources (Fin 1 → Fin 2 → Fin 3) ≡ (Σ _ : Fin 1 • Fin 2 × 1)
920 _ = refl
921
922 _ : sources (Σ f : (Fin 1 → Fin 2 → Fin 3 → Fin 4) • Fin 5)
923   ≡ (Σ f : (Fin 1 × Fin 2 × Fin 3) • Fin 5)
924 _ = refl
925
926 _ : ∀ {A B C : Set} → sources (A → B → C) ≡ (A × B × 1)
927 _ = refl
928
929 _ : ∀ {A B C D E : Set} → sources (A → B → C → D → E)
930   ≡ Σ A (λ _ → Σ B (λ _ → Σ C (λ _ → Σ D (λ _ → T))))
931 _ = refl

```

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

```

929 -- one implicit
930 _ : sources (∀ {x : N} → x ≡ x) ≡ 0
931

```

```

932   _ = refl
933
934   -- multiple implicits
935   _ : sources (∀ {x y z : ℕ} → x ≡ y) ≡ 0
936   _ = refl

```

The third stage can now be formed.

```

937   D3 = sources D2
938
939   3-sources : D3 ≡ λ (X : Set) → Σ z : 1 • Σ s : X • 0
940   3-sources = refl
941

```

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

```

942   {-# TERMINATING #-}
943   Σ→∪0 : Term → Term
944   Σ→∪0 (def (quote Σ) (h1 :: h0 :: arg i A :: arg i1 (lam v (abs s x)) :: []))
945     = def (quote ∪0) (h1 :: h0 :: arg i A :: vArg (Σ→∪0 (var-dec x)) :: [])
946   -- Interpret “End” in do-notation to be an empty, impossible, constructor.
947   Σ→∪0 (def (quote T) _) = def (quote ⊥) []
948   -- Walk under λ's and Π's.
949   Σ→∪0 (lam v (abs s x)) = lam v (abs s (Σ→∪0 x))
950   Σ→∪0 (Π [ x : A ] Bx) = Π [ x : A ] Σ→∪0 Bx
951   {-# CATCHALL #-}
952   Σ→∪0 t = t
953
954   macro
955     Σ→∪ : Term → Term → TC Unit.τ
956     Σ→∪ tm goal = normalise tm >>= m λ tm' → unify (Σ→∪0 tm') goal
957
958   -- Unit tests
959   _ : Σ→∪ (Π X : Set • (X → X)) ≡ (Π X : Set • (X → X)); _ = refl
960   _ : Σ→∪ (Π X : Set • Σ s : X • X) ≡ (Π X : Set • X ∪ X) ; _ = refl
961   _ : Σ→∪ (Π X : Set • Σ s : (X → X) • X) ≡ (Π X : Set • (X → X) ∪ X) ; _ = refl
962   _ : Σ→∪ (Π X : Set • Σ z : X • Σ s : (X → X) • T {ℓ0}) ≡ (Π X : Set • X ∪ (X → X) ∪ ⊥) ; _ = refl
963
964   D4 = Σ→∪ D3
965
966   4-unions : D4 ≡ λ X → 1 ∪ X ∪ 0
967   4-unions = refl
968

```

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

```

969   {-# NO_POSITIVITY_CHECK #-}
970   data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
971     μ : F (Fix F) → Fix F
972
973   D = Fix D4
974
975   -- Pattern synonyms for more compact presentation
976   pattern zeroD = μ (inj1 tt) -- : D
977   pattern sucD e = μ (inj2 (inj1 e)) -- : D → D
978
979   to : D → ℕ
980   to zeroD = 0
981   to (sucD x) = suc (to x)
982
983   from : ℕ → D
984   from zero = zeroD
985

```

```

981   from (suc n) = sucD (from n)
982
983   toofrom : ∀ n → to (from n) ≡ n
984   toofrom zero = refl
985   toofrom (suc n) = cong suc (toofrom n)
986
987   fromto : ∀ d → from (to d) ≡ d
988   fromto zeroD = refl
989   fromto (sucD x) = cong sucD (fromto x)

```

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

```

991   macro
992     termtype : Term → Term → TC Unit.T
993     termtype tm goal =
994       normalise tm
995       >>=ₘ λ tm' → unify goal (def (quote Fix) ((vArg (Σ→ℳ₀ (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”.

```

998   Inj₀ : ℕ → Term → Term
999   Inj₀ zero c = con (quote inj₁) (arg (arg-info visible relevant) c :: [])
1000   Inj₀ (suc n) c = con (quote inj₂) (vArg (Inj₀ n c) :: [])
1001
1002   -- Duality!
1003   -- i-th projection: proj₁ ∘ (proj₂ ∘ ... ∘ proj₂)
1004   -- i-th injection: (inj₂ ∘ ... ∘ inj₂) ∘ inj₁
1005
1006   macro
1007     Inj : ℕ → Term → Term → TC Unit.T
1008     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} :

```

1009   startD :  $\mathbb{D}$ 
1010   startD = Inj 0 (tt {ℓ₀})
1011
1012   nextD' :  $\mathbb{D} \rightarrow \mathbb{D}$ 
1013   nextD' d = Inj 1 d

```

A.12 :kind

```

1015   data Kind : Set where
1016     'record : Kind
1017     'typeclass : Kind
1018     'data : Kind
1019
1020   macro
1021     _:kind_ : Term → Term → Term → TC Unit.T
1022     _:kind_ t (con (quote 'record) _) goal = normalise (t app (quoteTerm 0))
1023     >>=ₘ λ t' → unify (waist-helper 0 t') goal
1024     _:kind_ t (con (quote 'typeclass) _) goal = normalise (t app (quoteTerm 1))
1025     >>=ₘ λ t' → unify (waist-helper 1 t') goal
1026     _:kind_ t (con (quote 'data) _) goal = normalise (t app (quoteTerm 1))
1027     >>=ₘ λ t' → normalise (waist-helper 1 t')
1028     >>=ₘ λ t'' → unify goal (def (quote Fix) ((vArg (Σ→ℳ₀ (sources₁ t'')))) :: []))
1029     _:kind_ t _ goal = unify t goal

```

Informally, `_:kind_` behaves as follows:

```

1030      C :kind 'record    = C :waist 0
1031      C :kind 'typeclass = C :waist 1
1032      C :kind 'data      = termtype (C :waist 1)

```

A.13 `termtype PointedSet` $\cong \mathbb{1}$

```

1034      -- termtype (PointedSet)  $\cong \top$  !
1035      One  : Context ( $\ell$ suc  $\ell_0$ )
1036      One  = do Carrier  $\leftarrow$  Set  $\ell_0$ 
1037             point   $\leftarrow$  Carrier
1038             End    { $\ell_0$ }
1039
1040      One : Set
1041      One = termtype (One :waist 1)
1042
1043      view1 : One  $\rightarrow \mathbb{1}$ 
1044      view1 emptyM = tt

```