

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 know ¹ 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 $Magma_0$ is the definition used in the library we are using, we are stuck with it, if we want to be compatible with other work.

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

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 5 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	$\prod X : \text{Set} \bullet \text{Set}$
DynamicSystem_2	$\prod X : \text{Set} \bullet \prod x : X \bullet \text{Set}$

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* $\prod X : \text{Set} \bullet \text{Set}$ 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.

```

153   id0 : Set1
154   id0 =  $\Pi X : \mathbf{Set} \bullet \Pi e : X \bullet X$ 
155
156   id1 :  $\Pi X : \mathbf{Set} \bullet \mathbf{Set}$ 
157   id1 =  $\lambda (X : \mathbf{Set}) \rightarrow \Pi e : X \bullet X$ 
158
159   id2 :  $\Pi X : \mathbf{Set} \bullet \Pi e : X \bullet \mathbf{Set}$ 
160   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 termtree 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 [Moggi 1991; ?]:

```

170   DynamicSystem : Context  $\ell_1$ 
171   DynamicSystem = do State  $\leftarrow \mathbf{Set}$ 
172                   start  $\leftarrow$  State
173                   next  $\leftarrow$  (State  $\rightarrow$  State)
174                   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 a product type, we expect there to be a recursive definition whose base case will be the essential identity of products, the unit type $\mathbb{1}$.

Table 1. Elaborations of DynamicSystem at various exposure levels

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	$\Pi \text{State} : \mathbf{Set} \bullet \Pi \text{start} : X \bullet \Pi \text{next} : \text{State} \rightarrow \text{State} \bullet \mathbb{1}$

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

```

190   {- “Contexts” are exposure-indexed types -}
191   Context =  $\lambda \ell \rightarrow \mathbb{N} \rightarrow \mathbf{Set} \ell$ 
192
193   {- Every type is a context -}
194   ‘_ :  $\forall \{\ell\} \rightarrow \mathbf{Set} \ell \rightarrow \text{Context} \ell$ 
195   ‘S =  $\lambda \_ \rightarrow S$ 

```

```

197   {- The “empty context” is the unit type -}
198   End :  $\forall \{\ell\} \rightarrow \text{Context } \ell$ 
199   End =  $\cdot \mathbb{1}$ 

```

It remains to identify the definition of the underlying bind operation $\gg=$. Classically, 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.

```

205   _>>=_ :  $\forall \{a \ b\}$ 
206            $\rightarrow (\Gamma : \text{Context } a)$ 
207            $\rightarrow (\forall \{n\} \rightarrow \Gamma \ n \rightarrow \text{Context } b)$ 
208            $\rightarrow \text{Context } (a \uplus b)$ 
209    $(\Gamma \gg= f) \text{ zero} = \sum \gamma : \Gamma \ 0 \bullet f \ \gamma \ 0$ 
210    $(\Gamma \gg= f) (\text{suc } n) = \prod \gamma : \Gamma \ n \bullet f \ \gamma \ n$ 

```

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.

```

215   ‘ Set >>=  $\lambda \text{ State} \rightarrow$  ‘ State >>=  $\lambda \text{ start} \rightarrow$  ‘ (State  $\rightarrow$  State) >>=  $\lambda \text{ next} \rightarrow$  End
216   {- or -}
217   do State  $\leftarrow$  ‘ Set
218       start  $\leftarrow$  ‘ State
219       next  $\leftarrow$  ‘ (State  $\rightarrow$  State)
220   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: With this definition, the above declaration `DynamicSystem` typechecks. However, `DynamicSystem i`

```

226   _>>=_ :  $\forall \{a \ b\}$ 
227            $\rightarrow (\Gamma : \text{Set } a) \text{ -- Main difference}$ 
228            $\rightarrow (\Gamma \rightarrow \text{Context } b)$ 
229            $\rightarrow \text{Context } (a \uplus b)$ 
230    $(\Gamma \gg= f) \text{ zero} = \sum \gamma : \Gamma \bullet f \ \gamma \ 0$ 
231    $(\Gamma \gg= f) (\text{suc } n) = \prod \gamma : \Gamma \bullet f \ \gamma \ n$ 

```

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

$\neq \text{DynamicSystem}_i$, instead `DynamicSystem i` are “factories”: Given i -many arguments, a product value is formed. What if we want to *instantiate* some of the factory arguments ahead of time?

```

238    $\mathcal{N}_0 : \text{DynamicSystem } 0 \quad \{- \text{ See the elaborations table above } -\}$ 
239    $\mathcal{N}_0 = \mathbb{N}, 0, \text{suc}, \text{tt}$ 
240
241    $\mathcal{N}_1 : \text{DynamicSystem } 1$ 
242    $\mathcal{N}_1 = \lambda \text{ State} \rightarrow ??? \{- \text{ Impossible to complete if “State” is empty! } -\}$ 
243
244   {- “Instantiating” X to be  $\mathbb{N}$  in “DynamicSystem 1” -}

```

```

246  $\mathcal{N}_1' : \text{let State} = \mathbb{N} \text{ in } \Sigma \text{ start} : \text{State} \bullet \Sigma s : (\text{State} \rightarrow \text{State}) \bullet \mathbb{1}$ 
247  $\mathcal{N}_1' = \emptyset, \text{ suc}, \text{ tt}$ 

```

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 not 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 macro 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,

```

259  $\Pi \rightarrow \lambda (\Pi \rightarrow \lambda (\text{DynamicSystem } 2))$ 
260  $\equiv \{- \text{Definition of DynamicSystem at exposure level 2 -}\}$ 
261  $\Pi \rightarrow \lambda (\Pi \rightarrow \lambda (\Pi X : \text{Set} \bullet \Pi s : X \bullet \Sigma n : X \rightarrow X \bullet \mathbb{1}))$ 
262  $\equiv \{- \text{Definition of } \Pi \rightarrow \lambda -\}$ 
263  $\Pi \rightarrow \lambda (\lambda X : \text{Set} \bullet \Pi s : X \bullet \Sigma n : X \rightarrow X \bullet \mathbb{1})$ 
264  $\equiv \{- \text{Homomorphism of } \Pi \rightarrow \lambda -\}$ 
265  $\lambda X : \text{Set} \bullet \Pi \rightarrow \lambda (\Pi s : X \bullet \Sigma n : X \rightarrow X \bullet \mathbb{1})$ 
266  $\equiv \{- \text{Definition of } \Pi \rightarrow \lambda -\}$ 
267  $\lambda X : \text{Set} \bullet \lambda s : X \bullet \Sigma n : X \rightarrow X \bullet \mathbb{1}$ 

```

For practicality, `_:waist_` is a macro 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.

$$\boxed{\begin{array}{l} \tau : \text{waist } n = \Pi \rightarrow \lambda^n (\tau \ n) \\ \text{-----} \\ f^0 \ x = x \\ \text{-----} \\ f^{n+1} \ x = f^n (f \ x) \end{array}}$$

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 `< . . . >`.

```

277 -- Expressions of the form “... , tt” may now be written “< ... >”
278 infixr 5 < _>
279 <> :  $\forall \{\ell\} \rightarrow \mathbb{1} \ \{\ell\}$ 
280 <> = tt
281
282 < :  $\forall \{\ell\} \ \{S : \text{Set } \ell\} \rightarrow S \rightarrow S$ 
283 < s = s
284
285 <_> :  $\forall \{\ell\} \ \{S : \text{Set } \ell\} \rightarrow S \rightarrow S \times (\mathbb{1} \ \{\ell\})$ 
286 s > = s , tt

```

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

```

290  $\mathcal{N}^0 : \text{DynamicSystem} : \text{waist } 0$ 
291  $\mathcal{N}^0 = \langle \mathbb{N}, \emptyset, \text{ suc} \rangle$ 
292
293  $\mathcal{N}^1 : (\text{DynamicSystem} : \text{waist } 1) \ \mathbb{N}$ 

```

```

295  $\mathcal{N}^1 = \langle \emptyset, \text{suc} \rangle$ 
296
297  $\mathcal{N}^2 : (\text{DynamicSystem} : \text{waist } 2) \mathbb{N} \emptyset$ 
298  $\mathcal{N}^2 = \langle \text{suc} \rangle$ 
299
300  $\mathcal{N}^3 : (\text{DynamicSystem} : \text{waist } 3) \mathbb{N} \emptyset \text{ suc}$ 
301  $\mathcal{N}^3 = \langle \rangle$ 

```

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) $\mathbb{N} \emptyset$` 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:

```

310 Monoid :  $\forall \ell \rightarrow \text{Context } (\ell \text{ suc } \ell)$ 
311 Monoid  $\ell = \text{do}$  Carrier  $\leftarrow \text{Set } \ell$ 
312   _ $\oplus$ _  $\leftarrow (\text{Carrier} \rightarrow \text{Carrier} \rightarrow \text{Carrier})$ 
313   Id  $\leftarrow \text{Carrier}$ 
314   leftId  $\leftarrow \forall \{x : \text{Carrier}\} \rightarrow x \oplus \text{Id} \equiv x$ 
315   rightId  $\leftarrow \forall \{x : \text{Carrier}\} \rightarrow \text{Id} \oplus x \equiv x$ 
316   assoc  $\leftarrow \forall \{x \ y \ z\} \rightarrow (x \oplus y) \oplus z \equiv x \oplus (y \oplus z)$ 
317   End  $\{\ell\}$ 
318

```

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. In particular, the description language \mathbb{D} for dynamical systems, below, declares concrete constructors for a certain fixpoint F ; i.e., $\mathbb{D} \cong \text{Fix } F$ where:

```

330 data  $\mathbb{D} : \text{Set where}$ 
331   startD :  $\mathbb{D}$ 
332   nextD :  $\mathbb{D} \rightarrow \mathbb{D}$ 
333
334 F : Set  $\rightarrow$  Set
335 F =  $\lambda (D : \text{Set}) \rightarrow 1 \uplus D$ 
336
337 data Fix (F : Set  $\rightarrow$  Set) : Set where
338    $\mu : F (\text{Fix } F) \rightarrow \text{Fix } F$ 
339

```

The problem is whether we can derive F from `DynamicSystem`. Let us attempt a quick calculation.

```

341 do X  $\leftarrow \text{Set}$ ; z  $\leftarrow X$ ; s  $\leftarrow (X \rightarrow X)$ ; End
342  $\Rightarrow \{- \text{ Use existing interpretation to obtain a record. -}\}$ 
343

```

```

344    $\Sigma X : \text{Set} \bullet \Sigma z : X \bullet \Sigma s : (X \rightarrow X) \bullet \mathbb{1}$ 
345    $\Rightarrow \{- \text{ Pull out the carrier, “:waist 1”, to obtain a type constructor using “}\Pi \rightarrow \lambda\text{”}$ 
346    $\lambda X : \text{Set} \bullet \Sigma z : X \bullet \Sigma s : (X \rightarrow X) \bullet \mathbb{1}$ 
347    $\Rightarrow \{- \text{ Termtypes constructors target the declared type, so only their sources matter}$ 
348    $\text{E.g., ‘}z : X\text{’ is a nullary constructor targeting the carrier ‘}X\text{’.$ 
349    $\text{This introduces } \mathbb{1} \text{ types, so any existing occurrences are dropped via } \mathbb{0}. -\}$ 
350    $\lambda X : \text{Set} \bullet \Sigma z : \mathbb{1} \bullet \Sigma s : X \bullet \mathbb{0}$ 
351    $\Rightarrow \{- \text{ Termtypes are sums of products. -}\}$ 
352    $\lambda X : \text{Set} \bullet \mathbb{1} \uplus X \uplus \mathbb{0}$ 
353    $\Rightarrow \{- \text{ Termtypes are fixpoints of type constructors. -}\}$ 
354    $\text{Fix } (\lambda X \bullet \mathbb{1} \uplus X) \text{ -- i.e., } \mathbb{D}$ 

```

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 , $\Sigma \rightarrow \uplus$, sources shown below can be found in appendices A.3.4, A.11.4, and A.11.3, respectively.

```

363    $\Downarrow \tau = \text{“reduce all de brujin indices within } \tau \text{ by 1”}$ 
364    $\Sigma \rightarrow \uplus (\Sigma a : A \bullet Ba) = A \uplus \Sigma \rightarrow \uplus (\Downarrow Ba)$ 
365   sources  $(\lambda x : (\Pi a : A \bullet Ba) \bullet \tau) = (\lambda x : A \bullet \text{sources } \tau)$ 
366   sources  $(\lambda x : A \bullet \tau) = (\lambda x : \mathbb{1} \bullet \text{sources } \tau)$ 
367   termtype  $\tau = \text{Fix } (\Sigma \rightarrow \uplus (\text{sources } \tau))$ 

```

It is instructive to visually see how \mathbb{D} is obtained from termtype in order to demonstrate that this approach to algebraic data types is practical.

```

372    $\mathbb{D} = \text{termtype } (\text{DynamicSystem} : \text{waist } 1)$ 
373
374   -- Pattern synonyms for more compact presentation
375   pattern startD =  $\mu$  (inj1 tt) -- :  $\mathbb{D}$ 
376   pattern nextD e =  $\mu$  (inj2 (inj1 e)) -- :  $\mathbb{D} \rightarrow \mathbb{D}$ 

```

With the pattern declarations, we can actually use these more meaningful names, 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:

```

380   to :  $\mathbb{D} \rightarrow \mathbb{N}$ 
381   to startD = 0
382   to (nextD x) = suc (to x)
383
384   from :  $\mathbb{N} \rightarrow \mathbb{D}$ 
385   from zero = startD
386   from (suc n) = nextD (from n)

```

Readers whose language does not have **pattern** clauses need not despair. With the macro $\text{Inj } n \ x = \mu \ (\text{inj}_2 \ ^n \ (\text{inj}_1 \ x))$, we may define $\text{startD} = \text{Inj } 0 \ \text{tt}$ and $\text{nextD } e = \text{Inj } 1 \ e$ —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 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 2019; 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 transpires. 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.

Table 2. Comparing the in-language Context mechanism with the PackageFormer editor extension

	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

The original PackageFormer paper 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:

Table 3. Contexts as a semantics for PackageFormer constructs

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

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 define it as a macro:

```

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 assign a full semantics to PackageFormer’s syntactic constructs due to Agda’s substantially weak 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 exponentially more effort due to the limited reflection mechanism and the intrusion of the stringent type system.

6 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 monads. These relationships are summarised in the following table.

Table 4. Contexts embody all kinds of grouping mechanisms

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”

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. 2019. 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, 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>
- Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda — A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17–20, 2009. Proceedings*. 73–78. https://doi.org/10.1007/978-3-642-03359-9_6
- 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
- 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.

7 VECTOR SPACES

```

VecSpc : Set → Context ℓ1
VecSpc F = do V ← Set
           0 ← F
           1 ← F
           _+_ ← (F → F → F)
           o ← V
           _*_ ← (F → V → V)
           _·_ ← (V → V → F)
           End0

AA : Set → Set → Set
AA F V = (VecSpc F :waist 1) V

BB : Set → Set
BB = λ X → termtype (VecSpc X :waist 1)
{-
Fix
  (λ γ →

```

```

540      T ⊔
541      T ⊔
542      Σ X (λ x → Σ X (λ x1 → T)) ⊔
543      T ⊔ Σ X (λ x → Σ γ (λ x1 → T)) ⊔ Σ γ (λ x → Σ γ (λ x1 → T)) ⊔ ⊥
544  -}
545
546  pattern 0s = μ (inj1 tt)
547  pattern 1s = μ (inj2 (inj1 tt))
548  pattern _+_ x y = μ (inj2 (inj2 (inj1 (x , (y , tt)))))
549  pattern 0v = μ (inj2 (inj2 (inj2 (inj1 tt))))
550  pattern _*_v x xs = μ (inj2 (inj2 (inj2 (inj2 (inj1 (x , (xs , tt)))))))
551  pattern _·v xs ys = μ (inj2 (inj2 (inj2 (inj2 (inj2 (inj1 (xs , (ys , tt))))))))
552
553  data Ring (Scalar : Set) : Set where
554    zeros : Ring Scalar
555    ones : Ring Scalar
556    pluss : Scalar → Scalar → Ring Scalar
557    zerov : Ring Scalar
558    prod : Scalar → Ring Scalar → Ring Scalar
559    dot : Ring Scalar → Ring Scalar → Ring Scalar
560
561  view : ∀ {X} → BB X → Ring X
562  view 0s = zeros
563  view 1s = ones
564  view (x +s y) = pluss x y
565  view 0v = zerov
566  view (x *v xs) = prod x (view xs)
567  view (xs ·v ys) = dot (view xs) (view ys)
568

```

8 OLD WHY SYNTAX

MAYBE_DELETE

The archetype for records and termtypes —algebraic data types— are monoids. They describe untyped compositional structures, such as programs in dynamically type-checked language. In turn, their termtype is linked lists which reify a monoid value —such as a program— as a sequence of values —i.e., a list of language instructions— which ‘evaluate’ to the original value. The shift to syntax gives rise to evaluators, optimisers, and constrained recursion-induction principles.

9 OLD GRAPH IDEAS

MAYBE_DELETE

9.1 From the old introduction section

For example, there are two ways to implement the type of graphs in the dependently-typed language Agda [Bove et al. 2009; Norell 2007]: Having the vertices be a parameter or having them be a field of the record. Then there is also the syntax for graph vertex relationships. Suppose a library designer decides to work with fully bundled graphs, Graph₀ below, then a user decides to write the function comap, which relabels the vertices of a graph, using a function f to transform vertices.

```

589 record Graph0 : Set1 where
590   constructor ⟨_,_⟩0
591   field
592     Vertex : Set
593     Edges : Vertex → Vertex → Set
594   comap0 : {A B : Set}
595     → (f : A → B)
596     → (Σ G : Graph0 • Vertex G ≡ B)
597     → (Σ H : Graph0 • Vertex H ≡ A)
598   comap0 {A} f (G , refl) = ⟨ A , (λ x y → Edges G (f x) (f y)) ⟩0 , refl

```

Since the vertices are packed away as components of the records, the only way for f to refer to them is to awkwardly refer to seemingly arbitrary types, only then to have the vertices of the input graph G and the output graph H be constrained to match the type of the relabelling function f . Without the constraints, we could not even write the function for Graph_0 . With such an importance, it is surprising to see that the occurrences of the constraint proofs are un insightful refl -exivity proofs.

What the user would really want is to unbundle Graph_0 at will, to expose the first argument, to obtain Graph_1 below. Then, in stark contrast, the implementation comap_1 does not carry any excesses baggage at the type level nor at the implementation level.

```

609 record Graph1 (Vertex : Set) : Set1 where
610   constructor ⟨_⟩1
611   field
612     Edges : Vertex → Vertex → Set
613
614   comap1 : {A B : Set}
615     → (f : A → B)
616     → Graph1 B
617     → Graph1 A
618   comap1 f ⟨ edges ⟩1 = ⟨ (λ x y → edges (f x) (f y)) ⟩1

```

With Graph_1 , one immediately sees that the comap operation “pulls back” the vertex type. Such an observation for Graph_0 is not as easy; requiring familiarity with quantifier laws such as the one-point rule and quantifier distributivity.

10 OLD FREE DATATYPES FROM THEORIES

MAYBE_DELETE

Astonishingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if $C : \text{Set} \rightarrow \text{Context } \ell_0$ then $C' = \lambda X \rightarrow \text{termtyp} (C X : \text{waist } 1)$ can be used to form ‘free, lawless, C -instances’. For instance, earlier we witnessed that the termtyp of dynamical systems is essentially the natural numbers.

Table 5. Data structures as free theories

Theory	Termtyp
Dynamical Systems	\mathbb{N}
Pointed Structures	Maybe
Monoids	Binary Trees

To obtain trees over some ‘value type’ Ξ , one must start at the theory of “monoids containing a given set Ξ ”. Similarly, by starting at “theories of pointed sets over a given set Ξ ”, the resulting

termtypes is the Maybe type constructor —another instructive exercise to the reader: Show that $\mathbb{P} \cong \text{Maybe}$.

```

638 PointedOver : Set → Context (lsuc ℓ₀)
639 PointedOver Ξ = do Carrier ← Set ℓ₀
640                  point  ← Carrier
641                  embed   ← (Ξ → Carrier)
642                  End
643
644
645
646 P : Set → Set
647 P X = termtypes (PointedOver X :waist 1)
648
649 -- Pattern synonyms for more compact presentation
650 pattern nothingP = μ (inj₁ tt)      -- : P
651 pattern justP e   = μ (inj₂ (inj₁ e)) -- : P → P

```

The final entry in the table is a well known correspondence, that we can, not only formally express, but also prove to be true. We present the setup and leave it as an instructive exercise to the reader to present a bijective pair of functions between \mathbb{M} and `TreeSkeleton`. Hint: Interactively case-split on values of \mathbb{M} until the declared patterns appear, then associate them with the constructors of `TreeSkeleton`.

```

658 M : Set
659 M = termtypes (Monoid ℓ₀ :waist 1)
660
661 -- Pattern synonyms for more compact presentation
662 pattern emptyM      = μ (inj₁ tt)      -- : M
663 pattern branchM l r = μ (inj₂ (inj₁ (l , r , tt))) -- : M → M → M
664 pattern absurdM a   = μ (inj₂ (inj₂ (inj₂ (inj₂ a)))) -- absurd values of 0
665
666 data TreeSkeleton : Set where
667   empty : TreeSkeleton
668   branch : TreeSkeleton → TreeSkeleton → TreeSkeleton

```

10.1 Collection Context

```

671 Collection : ∀ ℓ → Context (lsuc ℓ)
672 Collection ℓ = do
673   Elem   ← Set ℓ
674   Carrier ← Set ℓ
675   insert ← (Elem → Carrier → Carrier)
676   ∅       ← Carrier
677   isEmpty ← (Carrier → Bool)
678   insert-nonEmpty ← ∀ {e : Elem} {x : Carrier} → isEmpty (insert e x) ≡ false
679   End {ℓ}
680
681 ListColl : {ℓ : Level} → Collection ℓ 1
682 ListColl E = ⟨ List E
683               , _::_
684               , []
685               , (λ { [] → true; _ → false})

```

```

687         , (λ {x} {x = x1} → refl)
688     }
689
690     NCollection = (Collection ℓ0 :waist 2)
691         ("Elem"      = Digit)
692         ("Carrier"   = N)
693
694     --
695     -- i.e., (Collection ℓ0 :waist 2) Digit N
696
697     stack : NCollection
698     stack = { "insert"      = (λ d s → suc (10 * s + #→N d))
699         , "empty stack"    = 0
700         , "is-empty"      = (λ { 0 → true; _ → false})
701         -- Properties --
702         , (λ {d : Digit} {s : N} → refl {x = false})
703     }

```

10.2 Elem, Carrier, insert projections

```

705     Elem      : ∀ {ℓ} → Collection ℓ 0 → Set ℓ
706     Elem      = λ C → Field 0 C
707
708     Carrier   : ∀ {ℓ} → Collection ℓ 0 → Set ℓ
709     Carrier1 : ∀ {ℓ} → Collection ℓ 1 → (γ : Set ℓ) → Set ℓ
710     Carrier1' : ∀ {ℓ} {γ : Set ℓ} (C : (Collection ℓ :waist 1) γ) → Set ℓ
711
712     Carrier   = λ C → Field 1 C
713     Carrier1 = λ C γ → Field 0 (C γ)
714     Carrier1' = λ C → Field 0 C
715
716     insert    : ∀ {ℓ} (C : Collection ℓ 0) → (Elem C → Carrier C → Carrier C)
717     insert1  : ∀ {ℓ} (C : Collection ℓ 1) (γ : Set ℓ) → γ → Carrier1 C γ → Carrier C
718     insert1' : ∀ {ℓ} {γ : Set ℓ} (C : (Collection ℓ :waist 1) γ) → γ → Carrier1' C → Carrier C
719
720     insert    = λ C → Field 2 C
721     insert1  = λ C γ → Field 1 (C γ)
722     insert1' = λ C → Field 1 C
723
724     insert2  : ∀ {ℓ} (C : Collection ℓ 2) (El Cr : Set ℓ) → El → Cr → Cr
725     insert2' : ∀ {ℓ} {El Cr : Set ℓ} (C : (Collection ℓ :waist 2) El Cr) → El → Cr → Cr
726
727     insert2 = λ C El Cr → Field 0 (C El Cr)
728     insert2' = λ C → Field 0 C
729

```

11 OLD WHAT ABOUT THE META-LANGUAGE'S PARAMETERS? MAYBE_DELETE

Besides :waist, another way to introduce parameters into a context grouping mechanism is to use the language's existing utility of parameterising a context by another type —as was done earlier in PointedOver.

For example, a pointed set needn't necessarily be terminated with End.

```

736 PointedSet : Context  $\ell_1$ 
737 PointedSet = do Carrier  $\leftarrow$  Set
738             point  $\leftarrow$  Carrier
739             End { $\ell_1$ }

```

We instead form a grouping consisting of a single type and a value of that type, along with an instance of the parameter type Ξ .

```

743 PointedPF : ( $\Xi$  : Set1)  $\rightarrow$  Context  $\ell_1$ 
744 PointedPF  $\Xi$  = do Carrier  $\leftarrow$  Set
745             point  $\leftarrow$  Carrier
746             '  $\Xi$ 

```

Clearly $\text{PointedPF } \mathbb{1} \approx \text{PointedSet}$, so we have a more generic grouping mechanism. The natural next step is to consider other parameters such as PointedSet in-place of Ξ .

```

749 -- Convenience names
750 PointedSetr = PointedSet           :kind 'record
751 PointedPFr  =  $\lambda \Xi \rightarrow \text{PointedPF } \Xi$  :kind 'record
752
753 -- An extended record type: Two types with a point of each.
754 TwoPointedSets = PointedPFr PointedSetr,
755
756 _ : TwoPointedSets
757    $\equiv$  (  $\Sigma$  Carrier1 : Set •  $\Sigma$  point1 : Carrier1
758       •  $\Sigma$  Carrier2 : Set •  $\Sigma$  point2 : Carrier2 •  $\mathbb{1}$ )
759 _ = refl
760
761 -- Here's an instance
762 one : PointedSet :kind 'record
763 one =  $\mathbb{B}$  , false , tt
764
765 -- Another; a pointed natural extended by a pointed bool,
766 -- with particular choices for both.
767 two : TwoPointedSets
768 two =  $\mathbb{N}$  , 0 , one
769

```

More generally, *record structure can be dependent on values*:

```

771 _PointedSets :  $\mathbb{N} \rightarrow \text{Set}_1$ 
772 zero PointedSets =  $\mathbb{1}$ 
773 suc n PointedSets = PointedPFr (n PointedSets)
774
775 _ : 4 PointedSets
776    $\equiv$  (  $\Sigma$  Carrier1 : Set •  $\Sigma$  point1 : Carrier1
777       •  $\Sigma$  Carrier2 : Set •  $\Sigma$  point2 : Carrier2
778       •  $\Sigma$  Carrier3 : Set •  $\Sigma$  point3 : Carrier3
779       •  $\Sigma$  Carrier4 : Set •  $\Sigma$  point4 : Carrier4 •  $\mathbb{1}$ )
780 _ = refl
781

```

Using traditional grouping mechanisms, it is difficult to create the family of types $n \text{ PointedSets}$ since the number of fields, $2 \times n$, depends on n .

It is interesting to note that the termtype of `PointedPF` is the same as the termtype of `PointedOver`, the `Maybe` type constructor!

```

PointedD : (X : Set) → Set1
PointedD X = termtype (PointedPF (Lift _ X) :waist 1)

-- Pattern synonyms for more compact presentation
pattern nothingP = μ (inj1 tt)
pattern justP x   = μ (inj2 (lift x))

casingP : ∀ {X} (e : PointedD X)
          → (e ≡ nothingP) ⊔ (Σ x : X • e ≡ justP x)
casingP nothingP = inj1 refl
casingP (justP x) = inj2 (x , refl)

```

12 OLD NEXT STEPS

MAYBE_DELETE

We have shown how a bit of reflection allows us to have a compact, yet practical, one-stop-shop notation for records, typeclasses, and algebraic data types. There are a number of interesting directions to pursue:

- How to write a function working homogeneously over one variation and having it lift to other variations.
 - Recall the `comap` from the introductory section was written over `Graph :kind 'typeclass`; how could that particular implementation be massaged to work over `Graph :kind k` for any k .
- The current implementation for deriving termtypes presupposes only one carrier set positioned as the first entity in the grouping mechanism.
 - How do we handle multiple carriers or choose a carrier from an arbitrary position or by name? `PackageFormer` handles this by comparing names.
- How do we lift properties or invariants, simple \equiv -types that ‘define’ a previous entity to be top-level functions in their own right?

Lots to do, so little time.

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 Π and Σ and Products/Sums

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

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

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

1 = T {ℓ₀}
0 = ⊥

```

A.3 Reflection

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

```

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

```

A.3.1 Single argument application.

```

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

```

Notice that we maintain existing applications:

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

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

```

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

```

A.3.3 The Length of a Term.

```

883 arg-term : ∀ {ℓ} {A : Set ℓ} → (Term → A) → Arg Term → A
884 arg-term f (arg i x) = f x
885
886 {-# TERMINATING #-}
887 lengtht : Term → ℕ
888 lengtht (var x args)      = 1 + sum (List.map (arg-term lengtht) args)
889 lengtht (con c args)      = 1 + sum (List.map (arg-term lengtht) args)
890 lengtht (def f args)       = 1 + sum (List.map (arg-term lengtht) args)
891 lengtht (lam v (abs s x)) = 1 + lengtht x
892 lengtht (pat-lam cs args) = 1 + sum (List.map (arg-term lengtht) args)
893 lengtht (Π[ x : A ] Bx)    = 1 + lengtht Bx
894 {-# CATCHALL #-}
895 -- sort, lit, meta, unknown
896 lengtht t = 0

```

Here is an example use:

```

896 _ : lengtht (quoteTerm (Σ x : ℕ • x ≡ x)) ≡ 10
897 _ = 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 .

```

901 var-dec0 : (fuel : ℕ) → Term → Term
902 var-dec0 zero t = t
903 -- Let's use an "impossible" term.
904 var-dec0 (suc n) (var zero args)      = def (quote ⊥) []
905 var-dec0 (suc n) (var (suc x) args)    = var x args
906 var-dec0 (suc n) (con c args)          = con c (map-Args (var-dec0 n) args)
907 var-dec0 (suc n) (def f args)          = def f (map-Args (var-dec0 n) args)
908 var-dec0 (suc n) (lam v (abs s x))      = lam v (abs s (var-dec0 n x))
909 var-dec0 (suc n) (pat-lam cs args)      = pat-lam cs (map-Args (var-dec0 n) args)
910 var-dec0 (suc n) (Π[ s : arg i A ] B)    = Π[ s : arg i (var-dec0 n A) ] var-dec0 n B
911 {-# CATCHALL #-}
912 -- sort, lit, meta, unknown
913 var-dec0 n t = t

```

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

```

914 var-dec : Term → Term
915 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:

```

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

```

A.4 Context Monad

```

921 Context = λ ℓ → ℕ → Set ℓ
922
923 infix -1000 ' _
924 ' _ : ∀ {ℓ} → Set ℓ → Context ℓ
925 ' S = λ _ → S
926
927 End : ∀ {ℓ} → Context ℓ
928 End = ' T
929
930 End0 = End {ℓ0}
931

```

```

932  _>=>_ : ∀ {a b}
933        → (Γ : Set a) -- Main difference
934        → (Γ → Context b)
935        → Context (a ⊔ b)
936  (Γ >=> f) N.zero = Σ γ : Γ • f γ 0
937  (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 `⟨ . . . ⟩`.

```

941  -- Expressions of the form "... , tt" may now be written "⟨ ... ⟩"
942  infixr 5 ⟨ _⟩
943  ⟨⟩ : ∀ {ℓ} → T {ℓ}
944  ⟨⟩ = tt
945
946  ⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
947  ⟨ s = s
948
949  _⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × T {ℓ}
950  s ) = s , tt

```

A.6 DynamicSystem Context

```

951  DynamicSystem : Context (ℓsuc Level.zero)
952  DynamicSystem = do X ← Set
953                  z ← X
954                  s ← (X → X)
955                  End {Level.zero}
956
957  -- Records with n-Parameters, n : 0..3
958  A B C D : Set1
959  A = DynamicSystem 0 -- Σ X : Set • Σ z : X • Σ s : X → X • T
960  B = DynamicSystem 1 -- (X : Set) → Σ z : X • Σ s : X → X • T
961  C = DynamicSystem 2 -- (X : Set) (z : X) → Σ s : X → X • T
962  D = DynamicSystem 3 -- (X : Set) (z : X) → (s : X → X) → T
963
964  _ : A ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • T) ; _ = refl
965  _ : B ≡ (Π X : Set • Σ z : X • Σ s : (X → X) • T) ; _ = refl
966  _ : C ≡ (Π X : Set • Π z : X • Σ s : (X → X) • T) ; _ = refl
967  _ : D ≡ (Π X : Set • Π z : X • Π s : (X → X) • T) ; _ = refl
968
969  stability : ∀ {n} → DynamicSystem (3 + n)
970            ≡ DynamicSystem 3
971  stability = refl
972
973  B-is-empty : ¬ B
974  B-is-empty b = proj1( b ⊥ )
975
976  N0 : DynamicSystem 0
977  N0 = N , 0 , suc , tt
978
979  N : DynamicSystem 0
980  N = ⟨ N , 0 , suc ⟩
981
982  B-on-N : Set
983  B-on-N = let X = N in Σ z : X • Σ s : (X → X) • T

```

```

981   ex : B-on- $\mathbb{N}$ 
982   ex = ⟨ 0 , suc ⟩

```

A.7 $\Pi \rightarrow \lambda$

```

985    $\Pi \rightarrow \lambda$ -helper : Term  $\rightarrow$  Term
986    $\Pi \rightarrow \lambda$ -helper (pi a b)      = lam visible b
987    $\Pi \rightarrow \lambda$ -helper (lam a (abs x y)) = lam a (abs x ( $\Pi \rightarrow \lambda$ -helper y))
988   {-# CATCHALL #-}
989    $\Pi \rightarrow \lambda$ -helper x = x
990
991   macro
992      $\Pi \rightarrow \lambda$  : Term  $\rightarrow$  Term  $\rightarrow$  TC Unit.T
993      $\Pi \rightarrow \lambda$  tm goal = normalise tm >=>m  $\lambda$  tm'  $\rightarrow$  unify ( $\Pi \rightarrow \lambda$ -helper tm') goal

```

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

```

994   waist-helper :  $\mathbb{N} \rightarrow$  Term  $\rightarrow$  Term
995   waist-helper zero t      = t
996   waist-helper (suc n) t = waist-helper n ( $\Pi \rightarrow \lambda$ -helper t)
997
998   macro
999      $\_:\text{waist}_\_$  : Term  $\rightarrow$  Term  $\rightarrow$  Term  $\rightarrow$  TC Unit.T
1000      $\_:\text{waist}_\_$  t n goal = normalise (t app n)
1001                           >=>m  $\lambda$  t'  $\rightarrow$  unify (waist-helper (to $\mathbb{N}$  n) t') goal

```

A.9 DynamicSystem :waist i

```

1003   A' : Set1
1004   B' :  $\forall$  (X : Set)  $\rightarrow$  Set
1005   C' :  $\forall$  (X : Set) (x : X)  $\rightarrow$  Set
1006   D' :  $\forall$  (X : Set) (x : X) (s : X  $\rightarrow$  X)  $\rightarrow$  Set
1007
1008   A' = DynamicSystem :waist 0
1009   B' = DynamicSystem :waist 1
1010   C' = DynamicSystem :waist 2
1011   D' = DynamicSystem :waist 3
1012
1013    $\mathcal{N}^0$  : A'
1014    $\mathcal{N}^0$  = ⟨  $\mathbb{N}$  , 0 , suc ⟩
1015
1016    $\mathcal{N}^1$  : B'  $\mathbb{N}$ 
1017    $\mathcal{N}^1$  = ⟨ 0 , suc ⟩
1018
1019    $\mathcal{N}^2$  : C'  $\mathbb{N}$  0
1020    $\mathcal{N}^2$  = ⟨ suc ⟩
1021
1022    $\mathcal{N}^3$  : D'  $\mathbb{N}$  0 suc
1023    $\mathcal{N}^3$  = ⟨ ⟩

```

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

```

1024   _ : DynamicSystem 0  $\equiv$  DynamicSystem :waist 0
1025   _ = refl

```

A.10 Field projections

```

1026   Field0 :  $\mathbb{N} \rightarrow$  Term  $\rightarrow$  Term
1027   Field0 zero c      = def (quote proj1) (arg (arg-info visible relevant) c :: [])
1028   Field0 (suc n) c = Field0 n (def (quote proj2) (arg (arg-info visible relevant) c :: []))

```

```

macro
  Field :  $\mathbb{N} \rightarrow \text{Term} \rightarrow \text{Term} \rightarrow \text{TC Unit}.\top$ 
  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.

```

D1 = DynamicSystem 0

1-records : D1  $\equiv (\Sigma X : \text{Set} \bullet \Sigma z : X \bullet \Sigma s : (X \rightarrow X) \bullet \top)$ 
1-records = refl

```

A.11.2 Stage 2: Parameterised Records.

```

D2 = DynamicSystem :waist 1

2-funcs : D2  $\equiv (\lambda (X : \text{Set}) \rightarrow \Sigma z : X \bullet \Sigma s : (X \rightarrow X) \bullet \top)$ 
2-funcs = refl

```

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

```

_ :  $\forall \{x : \mathbb{N}\} \rightarrow \mathbb{N}$ 
   $\equiv \text{pi } (\text{arg } (\text{arg-info hidden relevant}) (\text{quoteTerm } \mathbb{N})) (\text{abs "x"} (\text{quoteTerm } \mathbb{N}))$ 
_ = refl

```

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

```

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

{-# TERMINATING #-}
sources1 : Term  $\rightarrow$  Term
sources1 ( $\Pi [a : \text{arg } (\text{arg-info hidden } \_) \ A] \ Ba$ ) = quoteTerm 0
sources1 ( $\Pi [a : \text{arg } i \ A] (\Pi [b : \text{arg } \_ \ Ba] \text{Cab})$ ) = def (quote _X_) (vArg A ::
  vArg (def (quote _X_) (vArg (var-dec Ba) :: vArg (var-dec (var-dec (sources0 Cab))) :: [])) :: [])
sources1 ( $\Pi [x : \text{arg } i \ A] \ Bx$ ) = A
sources1 (def (quote  $\Sigma$ ) ( $\ell_1 :: \ell_2 :: \tau :: \text{body}$ ))
  = def (quote  $\Sigma$ ) ( $\ell_1 :: \ell_2 :: \text{map-Arg sources}_0 \ \tau :: \text{List.map } (\text{map-Arg sources}_1) \ \text{body}$ )
-- This function introduces 1s, so let's drop any old occurrences a la 0.
sources1 (def (quote  $\top$ ) _) = def (quote 0) []
sources1 (lam v (abs s x)) = lam v (abs s (sources1 x))
sources1 (var x args) = var x (List.map (map-Arg sources1) args)
sources1 (con c args) = con c (List.map (map-Arg sources1) args)
sources1 (def f args) = def f (List.map (map-Arg sources1) args)
sources1 (pat-lam cs args) = pat-lam cs (List.map (map-Arg sources1) args)
{-# CATCHALL #-}
-- sort, lit, meta, unknown
sources1 t = t

```

We now form the macro and some unit tests.

```

macro
  sources : Term → Term → TC Unit.T
  sources tm goal = normalise tm >>=  $m$   $\lambda$  tm' → unify (sources1 tm') goal

_ : sources (N → Set) ≡ N
_ = refl

_ : sources ( $\Sigma$  x : (N → Fin 3) • N) ≡ ( $\Sigma$  x : N • N)
_ = refl

_ :  $\forall \{ \ell : \text{Level} \} \{ A B C : \text{Set} \}$ 
  → sources ( $\Sigma$  x : (A → B) • C) ≡ ( $\Sigma$  x : A • C)
_ = refl

_ : sources (Fin 1 → Fin 2 → Fin 3) ≡ ( $\Sigma$  _ : Fin 1 • Fin 2 × 1)
_ = refl

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

_ :  $\forall \{ A B C : \text{Set} \} \rightarrow$  sources (A → B → C) ≡ (A × B × 1)
_ = refl

_ :  $\forall \{ A B C D E : \text{Set} \} \rightarrow$  sources (A → B → C → D → E)
  ≡  $\Sigma$  A ( $\lambda$  _ →  $\Sigma$  B ( $\lambda$  _ →  $\Sigma$  C ( $\lambda$  _ →  $\Sigma$  D ( $\lambda$  _ → T))))
_ = refl

```

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

```

-- one implicit
_ : sources ( $\forall \{ x : N \} \rightarrow x \equiv x$ ) ≡ 0
_ = refl

-- multiple implicits
_ : sources ( $\forall \{ x y z : N \} \rightarrow x \equiv y$ ) ≡ 0
_ = refl

```

The third stage can now be formed.

```

D3 = sources D2

3-sources : D3 ≡  $\lambda$  (X : Set) →  $\Sigma$  z : 1 •  $\Sigma$  s : X • 0
3-sources = refl

```

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

```

{-# TERMINATING #-}
 $\Sigma \rightarrow \cup$  : Term → Term
 $\Sigma \rightarrow \cup$  (def (quote  $\Sigma$ ) (h1 :: h0 :: arg i A :: arg i1 (lam v (abs s x)) :: []))
  = def (quote  $\cup$ ) (h1 :: h0 :: arg i A :: vArg ( $\Sigma \rightarrow \cup$ ) (var-dec x)) :: []
-- Interpret “End” in do-notation to be an empty, impossible, constructor.
 $\Sigma \rightarrow \cup$  (def (quote T) _) = def (quote  $\perp$ ) []
-- Walk under  $\lambda$ 's and  $\Pi$ 's.
 $\Sigma \rightarrow \cup$  (lam v (abs s x)) = lam v (abs s ( $\Sigma \rightarrow \cup$  x))
 $\Sigma \rightarrow \cup$  ( $\Pi$  [ x : A ] Bx) =  $\Pi$  [ x : A ]  $\Sigma \rightarrow \cup$  Bx
{-# CATCHALL #-}
 $\Sigma \rightarrow \cup$  t = t

```

```

1128 macro
1129    $\Sigma \rightarrow \mathcal{U}$  : Term  $\rightarrow$  Term  $\rightarrow$  TC Unit.T
1130    $\Sigma \rightarrow \mathcal{U}$  tm goal = normalise tm >>=m  $\lambda$  tm'  $\rightarrow$  unify ( $\Sigma \rightarrow \mathcal{U}_0$  tm') goal
1131
1132 -- Unit tests
1133 _ :  $\Sigma \rightarrow \mathcal{U}$  ( $\prod X : \text{Set} \bullet (X \rightarrow X)$ )  $\equiv$  ( $\prod X : \text{Set} \bullet (X \rightarrow X)$ ); _ = refl
1134 _ :  $\Sigma \rightarrow \mathcal{U}$  ( $\prod X : \text{Set} \bullet \Sigma s : X \bullet X$ )  $\equiv$  ( $\prod X : \text{Set} \bullet X \mathcal{U} X$ ) ; _ = refl
1135 _ :  $\Sigma \rightarrow \mathcal{U}$  ( $\prod X : \text{Set} \bullet \Sigma s : (X \rightarrow X) \bullet X$ )  $\equiv$  ( $\prod X : \text{Set} \bullet (X \rightarrow X) \mathcal{U} X$ ) ; _ = refl
1136 _ :  $\Sigma \rightarrow \mathcal{U}$  ( $\prod X : \text{Set} \bullet \Sigma z : X \bullet \Sigma s : (X \rightarrow X) \bullet \top \{\ell_0\}$ )  $\equiv$  ( $\prod X : \text{Set} \bullet X \mathcal{U} (X \rightarrow X) \mathcal{U} \perp$ ) ; _ = refl
1137
1138 D4 =  $\Sigma \rightarrow \mathcal{U}$  D3
1139
1140 4-unions : D4  $\equiv$   $\lambda X \rightarrow \perp \mathcal{U} X \mathcal{U} \emptyset$ 
1141 4-unions = refl

```

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

```

1141 {-# NO_POSITIVITY_CHECK #-}
1142 data Fix {ℓ} (F : Set ℓ  $\rightarrow$  Set ℓ) : Set ℓ where
1143    $\mu$  : F (Fix F)  $\rightarrow$  Fix F
1144
1145  $\mathbb{D}$  = Fix D4
1146
1147 -- Pattern synonyms for more compact presentation
1148 pattern zeroD =  $\mu$  (inj1 tt) -- :  $\mathbb{D}$ 
1149 pattern sucD e =  $\mu$  (inj2 (inj1 e)) -- :  $\mathbb{D} \rightarrow \mathbb{D}$ 
1150
1151 to :  $\mathbb{D} \rightarrow \mathbb{N}$ 
1152 to zeroD = 0
1153 to (sucD x) = suc (to x)
1154
1155 from :  $\mathbb{N} \rightarrow \mathbb{D}$ 
1156 from zero = zeroD
1157 from (suc n) = sucD (from n)
1158
1159 toofrom :  $\forall n \rightarrow$  to (from n)  $\equiv$  n
1160 toofrom zero = refl
1161 toofrom (suc n) = cong suc (toofrom n)
1162
1163 fromto :  $\forall d \rightarrow$  from (to d)  $\equiv$  d
1164 fromto zeroD = refl
1165 fromto (sucD x) = cong sucD (fromto x)

```

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

```

1165 macro
1166   termtype : Term  $\rightarrow$  Term  $\rightarrow$  TC Unit.T
1167   termtype tm goal =
1168     normalise tm
1169     >>=m  $\lambda$  tm'  $\rightarrow$  unify goal (def (quote Fix) ((vArg ( $\Sigma \rightarrow \mathcal{U}_0$  (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”.

```

1171 Inj0 :  $\mathbb{N} \rightarrow$  Term  $\rightarrow$  Term
1172 Inj0 zero c = con (quote inj1) (arg (arg-info visible relevant) c :: [])
1173 Inj0 (suc n) c = con (quote inj2) (vArg (Inj0 n c) :: [])
1174
1175 -- Duality!
1176

```



```

1177 -- i-th projection: proj1 ∘ (proj2 ∘ ⋯ ∘ proj2)
1178 -- i-th injection: (inj2 ∘ ⋯ ∘ inj2) ∘ inj1
1179
1180 macro
1181   Inj : ℕ → Term → Term → TC Unit.T
1182   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} :

```

1183
1184 startD :  $\mathbb{D}$ 
1185 startD = Inj 0 (tt { $\ell_0$ })
1186
1187 nextD' :  $\mathbb{D} \rightarrow \mathbb{D}$ 
1188 nextD' d = Inj 1 d

```

A.12 Monoids

A.12.1 Context.

```

1191 Monoid :  $\forall \ell \rightarrow$  Context ( $\ell$ suc  $\ell$ )
1192 Monoid  $\ell$  = do Carrier  $\leftarrow$  Set  $\ell$ 
1193               Id  $\leftarrow$  Carrier
1194               _ $\oplus$ _  $\leftarrow$  (Carrier  $\rightarrow$  Carrier  $\rightarrow$  Carrier)
1195               leftId  $\leftarrow \forall \{x : \text{Carrier}\} \rightarrow x \oplus \text{Id} \equiv x$ 
1196               rightId  $\leftarrow \forall \{x : \text{Carrier}\} \rightarrow \text{Id} \oplus x \equiv x$ 
1197               assoc  $\leftarrow \forall \{x\ y\ z\} \rightarrow (x \oplus y) \oplus z \equiv x \oplus (y \oplus z)$ 
1198               End { $\ell$ }

```

A.12.2 Termtypes.

```

1200  $\mathbb{M} : \text{Set}$ 
1201  $\mathbb{M} = \text{termtyp} (\text{Monoid } \ell_0 : \text{waist } 1)$ 
1202 {- ie Fix ( $\lambda X \rightarrow \mathbb{1}$  -- Id, nil leaf
1203            $\cup X \times X \times \mathbb{1}$  --  $\_ \oplus \_$ , branch
1204            $\cup 0$  -- src of leftId
1205            $\cup 0$  -- src of rightId
1206            $\cup X \times X \times 0$  -- src of assoc
1207            $\cup 0$ ) -- the “End { $\ell$ }”
1208
1209 -- Pattern synonyms for more compact presentation
1210 pattern emptyM =  $\mu$  (inj1 tt) -- :  $\mathbb{M}$ 
1211 pattern branchM l r =  $\mu$  (inj2 (inj1 (l , r , tt))) -- :  $\mathbb{M} \rightarrow \mathbb{M} \rightarrow \mathbb{M}$ 
1212 pattern absurdM a =  $\mu$  (inj2 (inj2 (inj2 (inj2 a)))) -- absurd values of  $\mathbb{M}$ 
1213
1214 data TreeSkeleton : Set where
1215   empty : TreeSkeleton
1216   branch : TreeSkeleton  $\rightarrow$  TreeSkeleton  $\rightarrow$  TreeSkeleton

```

A.12.3 $\mathbb{M} \cong \text{TreeSkeleton}$.

```

1217  $\mathbb{M} \rightarrow \text{Tree} : \mathbb{M} \rightarrow \text{TreeSkeleton}$ 
1218  $\mathbb{M} \rightarrow \text{Tree}$  emptyM = empty
1219  $\mathbb{M} \rightarrow \text{Tree}$  (branchM l r) = branch ( $\mathbb{M} \rightarrow \text{Tree}$  l) ( $\mathbb{M} \rightarrow \text{Tree}$  r)
1220  $\mathbb{M} \rightarrow \text{Tree}$  (absurdM (inj1 ()))
1221  $\mathbb{M} \rightarrow \text{Tree}$  (absurdM (inj2 ()))
1222
1223  $\mathbb{M} \leftarrow \text{Tree} : \text{TreeSkeleton} \rightarrow \mathbb{M}$ 
1224  $\mathbb{M} \leftarrow \text{Tree}$  empty = emptyM
1225  $\mathbb{M} \leftarrow \text{Tree}$  (branch l r) = branchM ( $\mathbb{M} \leftarrow \text{Tree}$  l) ( $\mathbb{M} \leftarrow \text{Tree}$  r)

```

```

1226  $\mathbb{M} \leftarrow \text{Tree} \circ \mathbb{M} \rightarrow \text{Tree} : \forall m \rightarrow \mathbb{M} \leftarrow \text{Tree} (\mathbb{M} \rightarrow \text{Tree } m) \equiv m$ 
1227  $\mathbb{M} \leftarrow \text{Tree} \circ \mathbb{M} \rightarrow \text{Tree } \text{emptyM} = \text{refl}$ 
1228  $\mathbb{M} \leftarrow \text{Tree} \circ \mathbb{M} \rightarrow \text{Tree} (\text{branchM } l \ r) = \text{cong}_2 \ \text{branchM} \ (\mathbb{M} \leftarrow \text{Tree} \circ \mathbb{M} \rightarrow \text{Tree } l) \ (\mathbb{M} \leftarrow \text{Tree} \circ \mathbb{M} \rightarrow \text{Tree } r)$ 
1229  $\mathbb{M} \leftarrow \text{Tree} \circ \mathbb{M} \rightarrow \text{Tree} (\text{absurdM } (\text{inj}_1 \ \_))$ 
1230  $\mathbb{M} \leftarrow \text{Tree} \circ \mathbb{M} \rightarrow \text{Tree} (\text{absurdM } (\text{inj}_2 \ \_))$ 
1231
1232  $\mathbb{M} \rightarrow \text{Tree} \circ \mathbb{M} \leftarrow \text{Tree} : \forall t \rightarrow \mathbb{M} \rightarrow \text{Tree} (\mathbb{M} \leftarrow \text{Tree } t) \equiv t$ 
1233  $\mathbb{M} \rightarrow \text{Tree} \circ \mathbb{M} \leftarrow \text{Tree } \text{empty} = \text{refl}$ 
1234  $\mathbb{M} \rightarrow \text{Tree} \circ \mathbb{M} \leftarrow \text{Tree} (\text{branch } l \ r) = \text{cong}_2 \ \text{branch} \ (\mathbb{M} \rightarrow \text{Tree} \circ \mathbb{M} \leftarrow \text{Tree } l) \ (\mathbb{M} \rightarrow \text{Tree} \circ \mathbb{M} \leftarrow \text{Tree } r)$ 

```

A.13 :kind

```

1235 data Kind : Set where
1236   'record    : Kind
1237   'typeclass : Kind
1238   'data      : Kind
1239
1240 macro
1241   _ : kind_ : Term → Term → Term → TC Unit.T
1242   _ : kind_ t (con (quote 'record) _) goal = normalise (t app (quoteTerm 0))
1243                                     >>=  $_m \lambda t' \rightarrow \text{unify } (\text{waist-helper } 0 \ t') \ \text{goal}$ 
1244   _ : kind_ t (con (quote 'typeclass) _) goal = normalise (t app (quoteTerm 1))
1245                                     >>=  $_m \lambda t' \rightarrow \text{unify } (\text{waist-helper } 1 \ t') \ \text{goal}$ 
1246   _ : kind_ t (con (quote 'data) _) goal = normalise (t app (quoteTerm 1))
1247                                     >>=  $_m \lambda t' \rightarrow \text{normalise } (\text{waist-helper } 1 \ t')$ 
1248   _ : kind_ t _ goal = unify goal (def (quote Fix) ((vArg ( $\Sigma \rightarrow \mathbb{U}_0$  (sources1 t')))) :: []))

```

Informally, `_ : kind_` behaves as follows:

```

1249 C : kind 'record    = C : waist 0
1250 C : kind 'typeclass = C : waist 1
1251 C : kind 'data      = termtype (C : waist 1)

```

A.14 termtype PointedSet $\cong \mathbb{1}$

```

1254 -- termtype (PointedSet)  $\cong \mathbb{T} !$ 
1255 One : Context ( $\ell \text{ suc } \ell_0$ )
1256 One   = do Carrier  $\leftarrow$  Set  $\ell_0$ 
1257       point  $\leftarrow$  Carrier
1258       End { $\ell_0$ }
1259
1260 One : Set
1261 One = termtype (One : waist 1)
1262
1263 view1 : One  $\rightarrow \mathbb{1}$ 
1264 view1 emptyM = tt

```

A.15 The Termtyping of Graphs is Vertex Pairs

From simple graphs (relations) to a syntax about them: One describes a simple graph by presenting edges as pairs of vertices!

```

1267 PointedOver2 : Set  $\rightarrow$  Context ( $\ell \text{ suc } \ell_0$ )
1268 PointedOver2  $\Xi$  = do Carrier  $\leftarrow$  Set  $\ell_0$ 
1269               relation  $\leftarrow$  ( $\Xi \rightarrow \Xi \rightarrow$  Carrier)
1270               End { $\ell_0$ }
1271
1272  $\mathbb{P}_2 : \text{Set} \rightarrow \text{Set}$ 
1273  $\mathbb{P}_2 \ X = \text{termtype } (\text{PointedOver}_2 \ X : \text{waist } 1)$ 

```

```

1275     pattern _ $\rightleftharpoons$ _ x y =  $\mu$  (inj1 (x , y , tt))
1276
1277     view2 :  $\forall$  {X}  $\rightarrow$   $\mathbb{P}_2$  X  $\rightarrow$  X  $\times$  X
1278     view2 (x  $\rightleftharpoons$  y) = x , y

```

A.16 No ‘constants’, whence a type of infinitely branching terms

```

1280     PointedOver3 : Set  $\rightarrow$  Context ( $\ell_0$ )
1281     PointedOver3  $\Xi$  = do relation  $\leftarrow$  ( $\Xi \rightarrow \Xi \rightarrow \Xi$ )
1282                      End { $\ell_0$ }
1283
1284      $\mathbb{P}_3$  : Set
1285      $\mathbb{P}_3$  = termtype ( $\lambda$  X  $\rightarrow$  PointedOver3 X  $\emptyset$ )

```

A.17 \mathbb{P}_2 again!

```

1287     PointedOver4 : Context ( $\ell$ suc  $\ell_0$ )
1288     PointedOver4 = do  $\Xi \leftarrow$  Set
1289                      Carrier  $\leftarrow$  Set  $\ell_0$ 
1290                      relation  $\leftarrow$  ( $\Xi \rightarrow \Xi \rightarrow$  Carrier)
1291                      End { $\ell_0$ }
1292
1293     -- The current implementation of “termtype” only allows for one “Set” in the body.
1294     -- So we lift both out; thereby regaining  $\mathbb{P}_2$ !
1295
1296      $\mathbb{P}_4$  : Set  $\rightarrow$  Set
1297      $\mathbb{P}_4$  X = termtype ((PointedOver4 :waist 2) X)
1298
1299     pattern _ $\rightleftharpoons$ _ x y =  $\mu$  (inj1 (x , y , tt))
1300
1301     case4 :  $\forall$  {X}  $\rightarrow$   $\mathbb{P}_4$  X  $\rightarrow$  Set1
1302     case4 (x  $\rightleftharpoons$  y) = Set
1303
1304     -- Claim: Mention in paper.
1305     --
1306     -- P1 : Set  $\rightarrow$  Context =  $\lambda$   $\Xi \rightarrow$  do ... End
1307     --  $\cong$  P2 :waist 1
1308     -- where P2 : Context = do  $\Xi \leftarrow$  Set; ... End

```

A.18 \mathbb{P}_4 again – indexed unary algebras; i.e., “actions”

```

1309     PointedOver8 : Context ( $\ell$ suc  $\ell_0$ )
1310     PointedOver8 = do Index  $\leftarrow$  Set
1311                      Carrier  $\leftarrow$  Set
1312                      Operation  $\leftarrow$  (Index  $\rightarrow$  Carrier  $\rightarrow$  Carrier)
1313                      End { $\ell_0$ }
1314
1315      $\mathbb{P}_8$  : Set  $\rightarrow$  Set
1316      $\mathbb{P}_8$  X = termtype ((PointedOver8 :waist 2) X)
1317
1318     pattern _' x y =  $\mu$  (inj1 (x , y , tt))
1319
1320     view8 :  $\forall$  {I}  $\rightarrow$   $\mathbb{P}_8$  I  $\rightarrow$  Set1
1321     view8 (i  $\cdot$  e) = Set
1322
1323     **COMMENT Other experiments
1324     {- Yellow:
1325
1326     PointedOver5 : Context ( $\ell$ suc  $\ell_0$ )

```

```

1324 PointedOver5 = do One ← Set
1325                Two ← Set
1326                Three ← (One → Two → Set)
1327                End {ℓ0}
1328
1328  $\mathbb{P}_5 : \text{Set} \rightarrow \text{Set}_1$ 
1329  $\mathbb{P}_5 X = \text{termttype } ((\text{PointedOver}_5 : \text{waist } 2) X)$ 
1330 -- Fix ( $\lambda \text{ Two} \rightarrow \text{One} \times \text{Two}$ )
1331
1331 pattern  $\_::_5 \ x \ y = \mu \ (\text{inj}_1 \ (x, y, \text{tt}))$ 
1332
1333 case5 :  $\forall \{X\} \rightarrow \mathbb{P}_5 X \rightarrow \text{Set}_1$ 
1334 case5 (x ::5 xs) = Set
1335
1336 -}
1337
1338 -----
1339 {-- Dependent sums
1340
1341 PointedOver6 : Context ℓ1
1342 PointedOver6 = do Sort ← Set
1343                Carrier ← (Sort → Set)
1344                End {ℓ0}
1345
1346  $\mathbb{P}_6 : \text{Set}_1$ 
1347  $\mathbb{P}_6 = \text{termttype } ((\text{PointedOver}_6 : \text{waist } 1) )$ 
1348 -- Fix ( $\lambda X \rightarrow X$ )
1349
1350 -}
1351
1352 -----
1353 -- Distinuighed subset algebra
1354
1355 open import Data.Bool renaming (Bool to  $\mathbb{B}$ )
1356
1357 {-
1358 PointedOver7 : Context (ℓsuc ℓ0)
1359 PointedOver7 = do Index ← Set
1360                Is ← (Index →  $\mathbb{B}$ )
1361                End {ℓ0}
1362
1363 -- The current implementation of “termttype” only allows for one “Set” in the body.
1364 -- So we lift both out; thereby regaining  $\mathbb{P}_2$ !
1365
1366  $\mathbb{P}_7 : \text{Set} \rightarrow \text{Set}$ 
1367  $\mathbb{P}_7 X = \text{termttype } (\lambda \_ : \text{Set} \rightarrow (\text{PointedOver}_7 : \text{waist } 1) X)$ 
1368 --  $\mathbb{P}_1 X \cong X$ 
1369
1370 pattern  $\_ \rightleftharpoons \_ \ x \ y = \mu \ (\text{inj}_1 \ (x, y, \text{tt}))$ 
1371
1372 case7 :  $\forall \{X\} \rightarrow \mathbb{P}_7 X \rightarrow \text{Set}$ 
1373 case7 {X} (μ (inj1 x)) = X
1374
1375 -}

```

```

-----
{-
PointedOver9 : Context  $\ell_1$ 
PointedOver9      = do Carrier  $\leftarrow$  Set
                    End  $\{\ell_0\}$ 

-- The current implementation of “termtyping” only allows for one “Set” in the body.
-- So we lift both out; thereby regaining  $\mathbb{P}_2$ !

 $\mathbb{P}_9$  : Set
 $\mathbb{P}_9$  = termtyping ( $\lambda$  (X : Set)  $\rightarrow$  (PointedOver9 :waist 1) X)
--  $\cong \emptyset \cong \text{Fix } (\lambda X \rightarrow \emptyset)$ 
-}
```

A.19 Fix Id

```

PointedOver10 : Context  $\ell_1$ 
PointedOver10      = do Carrier  $\leftarrow$  Set
                    next       $\leftarrow$  (Carrier  $\rightarrow$  Carrier)
                    End  $\{\ell_0\}$ 

-- The current implementation of “termtyping” only allows for one “Set” in the body.
-- So we lift both out; thereby regaining  $\mathbb{P}_2$ !

 $\mathbb{P}_{10}$  : Set
 $\mathbb{P}_{10}$  = termtyping ( $\lambda$  (X : Set)  $\rightarrow$  (PointedOver10 :waist 1) X)
-- Fix ( $\lambda X \rightarrow X$ ), which does not exist.
```