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 $(M_1, \_ \mathbin{\mathring{\circ}}_1 \_, Id_1)$ and $(M_2, \_ \mathbin{\mathring{\circ}}_2 \_, Id_2)$, if we know [1] that $ceq : M_1 \equiv M_2$ then it is "obvious" that $Id_2 \mathbin{\mathring{\circ}}_2 (x \mathbin{\mathring{\circ}}_1 Id_1) \equiv x$ for all $x : M_1$. However, as written, this does not type-check. This is because $\_ \mathbin{\mathring{\circ}}_2 \_$ 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 artifically 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:

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


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

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

This is the 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.

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

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

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

So not only are there no general rules for when to bundle or not, it is in fact guaranteed that any given choice will be sub-optimal for certain applications. Furthermore, these types are equivalent, as we can "pack away" an exposed piece, e.g., $\text{Monoid}_0 \cong \Sigma\ M : \textbf{Set} \bullet \text{Monoid}_1\ M$. 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, _: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
$DynamicSystem_1$	Π X : Set • Set
$DynamicSystem_2$	Π X : Set • Π x : X • Set

We shall refer to the concern of moving from a record to a parameterised record as **the unbundling problem** [Garillot et al. 2009]. For example, moving from the *type* Set_1 to the *function type* Π X : **Set** • **Set** gets us from $DynamicSystem_0$ to something resembling $DynamicSystem_1$, which we arrive at if we can obtain a *type constructor* λ X : **Set** • ⋯. We shall refer to the latter change as *reïfication* since the result is more concrete: It can be applied. This transformation will be denoted by $\Pi \to \lambda$. 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: $id_{i+1} = \Pi{\rightarrow}\lambda\ id_i$ The latter identity is proven by reflexivity in the appendices.

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

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

id₂ : Π X : Set • Π e : X • Set
id₂ = λ (X : Set) (e : X) → X
```

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

3 MONADIC NOTATION

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

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

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

Table 1. Elaborations of DynamicSystem at various exposure levels

Exposure	Elaboration
0	Σ State : Set • Σ start : X • Σ next : State → State • $\mathbb{1}$
1	Π State : Set • Σ start : X • Σ next : State → State • $\mathbb{1}$
2	Π State : Set • Π start : X • Σ next : State → State • $\mathbb{1}$
3	Π State : Set • Π start : X • Π next : State → State • $\mathbb{1}$

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

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

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

```
197        ‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
198        ‘ S = λ _ → S
199
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 >>=. Usually, for a type constructor m, bind is typed ∀ {X Y : Set} → m X → (X → m Y) → m Y. It allows one to "extract an X-value for later use" in the m Y context. Since our m = Context is from levels to types, we need to slightly alter bind's typing.

```
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, >>=, 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 ≇ 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?

```
240        𝒩_0 : DynamicSystem 0    {- See the elaborations in Table 1 -}
241        𝒩_0 = ℕ , 0 , suc , tt
242
243        𝒩_1 : DynamicSystem 1
244        𝒩_1 = λ State → ??? {- Impossible to complete if "State" is empty! -}
245
```

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

It seems what we need is a method, say $\Pi\to\lambda$, 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 $\Pi\to\lambda$. 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 $\Pi\to\lambda$ is thus a macro [2] that acts on the syntactic term representations of types. Below is main transformation —the details can be found in Appendix A.7.

$$\Pi\to\lambda \ (\Pi \ a : A \ ● \ \tau) = (\lambda \ a : A \ ● \ \tau)$$

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

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

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

$$\tau \ \text{:waist } n = \Pi\to\lambda^n \ (\tau \ n)$$
$$f^0 \ x = x$$
$$f^{n+1} \ x = f^n \ (f \ x)$$

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

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

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

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

---

[2]A *macro* is a function that manipulates the abstract syntax trees of the host language. In particular, it may take an arbitrary term, shuffle its syntax to provide possibly meaningless terms or terms that could not be formed without pattern matching on the possible syntactic constructions.

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

```
𝒩⁰ : DynamicSystem :waist 0
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : (DynamicSystem :waist 1) ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : (DynamicSystem :waist 2) ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : (DynamicSystem :waist 3) ℕ 0 suc
𝒩³ = ⟨⟩
```

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

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

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              _⊕_    ← (Carrier → Carrier → Carrier)
              Id     ← Carrier
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

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

4 TERMTYPES AS FIXED-POINTS

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

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

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

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

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

```
do X ← Set; z ← X; s ← (X → X); End
⇒ {- Use existing interpretation to obtain a record. -}
Σ X : Set • Σ z : X • Σ s : (X → X) • 𝟙
⇒ {- Pull out the carrier, ":waist 1", to obtain a type constructor using "Π→λ"
λ X : Set • Σ z : X • Σ s : (X → X) • 𝟙
⇒ {- Termtype constructors target the declared type, so only their sources matte
    E.g., 'z : X' is a nullary constructor targeting the carrier 'X'.
    This introduces 𝟙 types, so any existing occurances are dropped via 𝟘. -}
λ X : Set • Σ z : 𝟙 • Σ s : X • 𝟘
⇒ {- Termtypes are sums of products. -}
λ X : Set •        𝟙    ⊎    X    ⊎ 𝟘
⇒ {- Termtypes are fixpoints of type constructors. -}
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 op : A → B targets the type termtype B and so its source is A. The details on the operations ⇊, $\Sigma{\to}\uplus$, sources shown below can be found in appendices A.3.4, A.11.4, and A.11.3, respectively.

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

Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (⇊ Ba)

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

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

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

```
𝔻 = termtype (DynamicSystem :waist 1)

-- Pattern synonyms for more compact presentation
pattern startD  = μ (inj₁ tt)          -- : 𝔻
pattern nextD e = μ (inj₂ (inj₁ e)) -- : 𝔻 → 𝔻
```

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

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

from : ℕ → 𝔻
```

```
        from zero     = startD
        from (suc n) = nextD (from n)
```

Readers whose language does not have **pattern** clauses need not despair. With the macro $\overline{\text{Inj n x} = \mu \ (\text{inj}_2 \ ^n \ (\text{inj}_1 \ x))}$, 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 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
─⊕→	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	do S ← Set; s ← S; n ← (S → S); End	"name-type pairs"
Record Type	Σ S : Set • Σ s : S • Σ n : S → S • $\mathbb{1}$	"bundled-up data"
Function Type	Π S • Σ s : S • Σ n : S → S • $\mathbb{1}$	"a type of functions"
Type constructor	λ S • Σ s : S • Σ n : S → S • $\mathbb{1}$	"a function on types"
Algebraic datatype	data $\mathbb{D}$: Set where s : $\mathbb{D}$; n : $\mathbb{D}$ → $\mathbb{D}$	"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

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
```

```
540        AA : Set → Set → Set
541        AA F V = (VecSpc F :waist 1) V
542
543        BB : Set → Set
544        BB = λ X → termtype (VecSpc X :waist 1)
545        {-
546        Fix
547            (λ γ →
548                ⊤ ⊎
549                ⊤ ⊎
550                Σ X (λ x → Σ X (λ x₁ → ⊤)) ⊎
551                ⊤ ⊎ Σ X (λ x → Σ γ (λ x₁ → ⊤)) ⊎ Σ γ (λ x → Σ γ (λ x₁ → ⊤)) ⊎ ⊥)
552        -}
553
554        pattern 𝟘ₛ = μ (inj₁ tt)
555        pattern 𝟙ₛ = μ (inj₂ (inj₁ tt))
556        pattern _+ₛ_ x y = μ (inj₂ (inj₂ (inj₁ (x , (y , tt)))))
557        pattern 𝟘ᵥ = μ (inj₂ (inj₂ (inj₂ (inj₁ tt))))
558        pattern _*ᵥ_ x xs = μ (inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (x , (xs , tt)))))))
559        pattern _·ᵥ_ xs ys = μ (inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (xs , (ys , tt))))))))
560
561        data ℝ𝕚𝕟𝕘 (Scalar : Set) : Set where
562          zeroₛ : ℝ𝕚𝕟𝕘 Scalar
563          oneₛ  : ℝ𝕚𝕟𝕘 Scalar
564          plusₛ : Scalar → Scalar → ℝ𝕚𝕟𝕘 Scalar
565          zeroᵥ : ℝ𝕚𝕟𝕘 Scalar
566          prod  : Scalar → ℝ𝕚𝕟𝕘 Scalar → ℝ𝕚𝕟𝕘 Scalar
567          dot   : ℝ𝕚𝕟𝕘 Scalar → ℝ𝕚𝕟𝕘 Scalar → ℝ𝕚𝕟𝕘 Scalar
568
569        view : ∀ {X} → BB X → ℝ𝕚𝕟𝕘 X
570        view 𝟘ₛ = zeroₛ
571        view 𝟙ₛ = oneₛ
572        view (x +ₛ y) = plusₛ x y
573        view 𝟘ᵥ = zeroᵥ
574        view (x *ᵥ xs) = prod x (view xs)
575        view (xs ·ᵥ ys) = dot (view xs) (view ys)
```

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, $Graph_0$ below, then a user decides to write the function comap, which relabels the vertices of a graph, using a function f to transform vertices.

```
         record Graph₀ : Set₁ where
           constructor ⟨_,_⟩₀
           field
             Vertex : Set
             Edges : Vertex → Vertex → Set
         comap₀ : {A B : Set}
                 → (f : A → B)
                 → (Σ G : Graph₀ • Vertex G ≡ B)
                 → (Σ H : Graph₀ • Vertex H ≡ A)
         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 Graph₀. With such an importance, it is surprising to see that the occurrences of the constraint proofs are uninsightful refl-exivity proofs.

What the user would really want is to unbundle Graph₀ at will, to expose the first argument, to obtain Graph₁ below. Then, in stark contrast, the implementation comap₁ does not carry any excesses baggage at the type level nor at the implementation level.

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


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

With Graph₁, one immediately sees that the comap operation "pulls back" the vertex type. Such an observation for Graph₀ 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 `: Set` → `Context` ℓ_0 then $\mathbb{C}$' `=` λ X → `termtype` (C X `:waist` `1`) can be used to form 'free, lawless, C-instances'. For instance, earlier we witnessed that the termtype of dynamical systems is essentially the natural numbers.

Table 5. Data structures as free theories

Theory	Termtype
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

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

```
PointedOver  : Set → Context (ℓsuc ℓ₀)
PointedOver Ξ    = do Carrier ← Set ℓ₀
                      point   ← Carrier
                      embed   ← (Ξ → Carrier)
                      End


ℙ : Set → Set
ℙ X = termtype (PointedOver X :waist 1)


-- Pattern synonyms for more compact presentation
pattern nothingP = μ (inj₁ tt)         -- : ℙ
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.

```
𝕄 : Set
𝕄 = termtype (Monoid ℓ₀ :waist 1)


-- Pattern synonyms for more compact presentation
pattern emptyM       = μ (inj₁ tt)                      -- : 𝕄
pattern branchM l r = μ (inj₂ (inj₁ (l , r , tt)))     -- : 𝕄 → 𝕄 → 𝕄
pattern absurdM a   = μ (inj₂ (inj₂ (inj₂ (inj₂ a))))  -- absurd values of 𝕆


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

10.1 Collection Context

```
Collection : ∀ ℓ → Context (ℓsuc ℓ)
Collection ℓ = do
  Elem    ← Set ℓ
  Carrier ← Set ℓ
  insert  ← (Elem → Carrier → Carrier)
  ∅       ← Carrier
  isEmpty ← (Carrier → Bool)
  insert-nonEmpty ← ∀ {e : Elem} {x : Carrier} → isEmpty (insert e x) ≡ false
  End {ℓ}


ListColl : {ℓ : Level} → Collection ℓ 1
ListColl E = ⟨ List E
             , _∷_
             , []
             , (λ { [] → true; _ → false})
```

```
                        , (λ {x} {x = x₁} → refl)
                        ⟩

        ℕCollection = (Collection ℓ₀ :waist 2)
                        ("Elem"    ≔ Digit)
                        ("Carrier" ≔ ℕ)
        --
        -- i.e., (Collection ℓ₀ :waist 2) Digit ℕ

        stack : ℕCollection
        stack = ⟨ "insert"      ≔ (λ d s → suc (10 * s + #→ℕ d))
                , "empty stack" ≔ 0
                , "is-empty"    ≔ (λ { 0 → true; _ → false})
                -- Properties --
                , (λ {d : Digit} {s : ℕ} → refl {x = false})
                ⟩
```

10.2   Elem, Carrier, insert projections

```
        Elem       : ∀ {ℓ} → Collection ℓ 0 → Set ℓ
        Elem       = λ C   → Field 0 C

        Carrier   : ∀ {ℓ} → Collection ℓ 0 → Set ℓ
        Carrier₁  : ∀ {ℓ} → Collection ℓ 1 → (γ : Set ℓ) → Set ℓ
        Carrier₁' : ∀ {ℓ} {γ : Set ℓ} (C : (Collection ℓ :waist 1) γ) → Set ℓ

        Carrier   = λ C   → Field 1 C
        Carrier₁  = λ C γ → Field 0 (C γ)
        Carrier₁' = λ C   → Field 0 C

        insert    : ∀ {ℓ} (C : Collection ℓ 0) → (Elem C → Carrier C → Carrier C)
        insert₁   : ∀ {ℓ} (C : Collection ℓ 1) (γ : Set ℓ) →  γ → Carrier₁ C γ → Carrier
        insert₁'  : ∀ {ℓ} {γ : Set ℓ} (C : (Collection ℓ :waist 1) γ) → γ → Carrier₁' C →

        insert    = λ C   → Field 2 C
        insert₁   = λ C γ → Field 1 (C γ)
        insert₁'  = λ C   → Field 1 C

        insert₂   : ∀ {ℓ} (C : Collection ℓ 2) (El Cr : Set ℓ) → El → Cr → Cr
        insert₂'  : ∀ {ℓ} {El Cr : Set ℓ} (C : (Collection ℓ :waist 2) El Cr) → El → Cr →

        insert₂  = λ C El Cr → Field 0 (C El Cr)
        insert₂' = λ C → Field 0 C
```

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 termed with End.

```
785        PointedSet : Context ℓ₁
786        PointedSet = do Carrier ← Set
787                        point   ← Carrier
788                        End {ℓ₁}
```

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.

```
792        PointedPF : (Ξ : Set₁) → Context ℓ₁
793        PointedPF Ξ = do Carrier ← Set
794                          point   ← Carrier
795                          ‵ Ξ
```

Clearly PointedPF $\mathbb{1}$ ≈ 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.

```
799        -- Convenience names
800        PointedSet_r = PointedSet          :kind ‵record
801        PointedPF_r  = λ Ξ → PointedPF Ξ :kind ‵record
802
803        -- An extended record type: Two types with a point of each.
804        TwoPointedSets = PointedPF_r PointedSet_r
805
806        _ :   TwoPointedSets
807          ≡ ( Σ Carrier₁ : Set • Σ point₁ : Carrier₁
808            • Σ Carrier₂ : Set • Σ point₂ : Carrier₂ • 𝟙)
809        _ = refl
810
811        -- Here's an instance
812        one : PointedSet :kind ‵record
813        one = 𝔹 , false , tt
814
815        -- Another; a pointed natural extended by a pointed bool,
816        -- with particular choices for both.
817        two : TwoPointedSets
818        two = ℕ , 0 , one
```

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

```
820        _PointedSets : ℕ → Set₁
821        zero  PointedSets = 𝟙
822        suc n PointedSets = PointedPF_r (n PointedSets)
823
824
825        _ :   4 PointedSets
826          ≡ (Σ Carrier₁ : Set • Σ point₁ : Carrier₁
827            • Σ Carrier₂ : Set • Σ point₂ : Carrier₂
828            • Σ Carrier₃ : Set • Σ point₃ : Carrier₃
829            • Σ Carrier₄ : Set • Σ point₄ : Carrier₄ • 𝟙)
830        _ = refl
```

Using traditional grouping mechanisms, it is difficult to create the family of types n 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 ≡-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:•/Σ:• and Products/Sums

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

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

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

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

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

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

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

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

record ⊤ {ℓ} : Set ℓ where
  constructor tt

𝟙 = ⊤ {ℓ₀}
𝟘 = ⊥
```

A.3 Reflection

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

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

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:

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

A.3.2 Reify ℕ term encodings as ℕ values.

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

A.3.3 The Length of a Term.

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

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

Here is an example use:

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

A.3.4 Decreasing de Brujin Indices. Given a quantification (⊕ x : τ • fx), its body fx may refer to a free variable x. If we decrement all de Brujin indices fx contains, then there would be no reference to x.

```
var-dec_0 : (fuel : ℕ) → Term → Term
var-dec_0 zero t = t
-- Let's use an "impossible" term.
var-dec_0 (suc n) (var zero args)       = def (quote ⊥) []
var-dec_0 (suc n) (var (suc x) args)    = var x args
var-dec_0 (suc n) (con c args)          = con c (map-Args (var-dec_0 n) args)
var-dec_0 (suc n) (def f args)          = def f (map-Args (var-dec_0 n) args)
var-dec_0 (suc n) (lam v (abs s x))     = lam v (abs s (var-dec_0 n x))
var-dec_0 (suc n) (pat-lam cs args)     = pat-lam cs (map-Args (var-dec_0 n) args)
var-dec_0 (suc n) (Π[ s : arg i A ] B) = Π[ s : arg i (var-dec_0 n A) ] var-dec_0 n B
{-# CATCHALL #-}
-- sort, lit, meta, unknown
var-dec_0 n t = t
```

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

```
var-dec : Term → Term
var-dec t = var-dec_0 (length_t t) t
```

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

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

A.4 Context Monad

```
Context = λ ℓ → ℕ → Set ℓ

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

End : ∀ {ℓ} → Context ℓ
End = ` ⊤

End_0 = End {ℓ_0}
```

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

A.5 ⟨⟩ Notation

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

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

⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
⟨ s = s

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

A.6   DynamicSystem Context

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

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

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

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

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

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

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

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

```
1030          ex : B-on-ℕ
1031          ex = ⟨ 0 , suc ⟩
1032
```

A.7   Π→λ

```
1034          Π→λ-helper : Term → Term
1035          Π→λ-helper (pi   a b)        = lam visible b
1036          Π→λ-helper (lam a (abs x y)) = lam a (abs x (Π→λ-helper y))
1037          {-# CATCHALL #-}
1037          Π→λ-helper x = x
1038
1039          macro
1040            Π→λ : Term → Term → TC Unit.⊤
1040            Π→λ tm goal = normalise tm >>=ₘ λ tm' → unify (Π→λ-helper tm') goal
1041
```

A.8   _:waist_

```
1043          waist-helper : ℕ → Term → Term
1044          waist-helper zero t    = t
1045          waist-helper (suc n) t = waist-helper n (Π→λ-helper t)
1046
1047          macro
1048            _:waist_ : Term → Term → Term → TC Unit.⊤
1049            _:waist_ t n goal =      normalise (t app n)
1049                               >>=ₘ λ t' → unify (waist-helper (toℕ n) t') goal
1050
```

A.9   DynamicSystem :waist i

```
1052          A' : Set₁
1053          B' : ∀ (X : Set) → Set
1054          C' : ∀ (X : Set) (x : X) → Set
1055          D' : ∀ (X : Set) (x : X) (s : X → X) → Set
1055
1056          A' = DynamicSystem :waist 0
1057          B' = DynamicSystem :waist 1
1058          C' = DynamicSystem :waist 2
1059          D' = DynamicSystem :waist 3
1060
1060          𝒩⁰ : A'
1061          𝒩⁰ = ⟨ ℕ , 0 , suc ⟩
1062
1063          𝒩¹ : B' ℕ
1064          𝒩¹ = ⟨ 0 , suc ⟩
1065
1065          𝒩² : C' ℕ 0
1066          𝒩² = ⟨ suc ⟩
1067
1068          𝒩³ : D' ℕ 0 suc
1069          𝒩³ = ⟨⟩
```

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

```
1071          _ : DynamicSystem 0 ≡ DynamicSystem :waist 0
1072          _ = refl
1073
```

A.10   Field projections

```
1075          Field₀ : ℕ → Term → Term
1076          Field₀ zero c    = def (quote proj₁) (arg (arg-info visible relevant) c :: [])
1076          Field₀ (suc n) c = Field₀ n (def (quote proj₂) (arg (arg-info visible relevant) c :: []))
1077
1078
```

```
macro
  Field : ℕ → Term → Term → TC Unit.⊤
  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 ≡ (Σ X : Set • Σ z : X • Σ s : (X → X) • ⊤)
1-records = refl
```

A.11.2 Stage 2: Parameterised Records.

```
D₂ = DynamicSystem :waist 1

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

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

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

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

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


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

We now form the macro and some unit tests.

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

_ : sources (ℕ → Set) ≡ ℕ
_ = refl

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

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

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

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

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

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

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

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

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

The third stage can now be formed.

```
D₃ = sources D₂

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

A.11.4 *Stage 4: Σ→⊎ –Replacing Products with Sums.*

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

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

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

D₄ = Σ→⊎ D₃

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

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

```
{-# NO_POSITIVITY_CHECK #-}
data Fix {ℓ} (F : Set ℓ → Set ℓ) : Set ℓ where
  μ : F (Fix F) → Fix F

𝔻 = Fix D₄

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

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

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

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

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

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

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

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

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

-- Duality!
```

```
1226          -- i-th projection: proj₁ ∘ (proj₂ ∘ ⋯ ∘ proj₂)
1227          -- i-th injection:  (inj₂ ∘ ⋯ ∘ inj₂) ∘ inj₁
1228
1229          macro
               Inj : ℕ → Term → Term → TC Unit.⊤
1230           Inj n t goal = unify goal ((con (quote μ) []) app (Inj₀ n t))
```

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

```
1232          startD : 𝔻
1233          startD = Inj 0 (tt {ℓ₀})
1234
1235          nextD' : 𝔻 → 𝔻
1236          nextD' d = Inj 1 d
```

A.12 Monoids

A.12.1 Context.

```
1240          Monoid : ∀ ℓ → Context (ℓsuc ℓ)
1241          Monoid ℓ = do Carrier ← Set ℓ
1242                        Id       ← Carrier
1243                        _⊕_      ← (Carrier → Carrier → Carrier)
1244                        leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
1245                        rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
                            assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
1246                        End {ℓ}
```

A.12.2 Termtypes.

```
1249          𝕄 : Set
1250          𝕄 = termtype (Monoid ℓ₀ :waist 1)
              {- ie Fix (λ X → 𝟙            -- Id, nil leaf
1251                       ⊎ X × X × 𝟙 -- _⊕_, branch
1252                       ⊎ 𝟘          -- src of leftId
1253                       ⊎ 𝟘          -- src of rightId
1254                       ⊎ X × X × 𝟘 -- src of assoc
                          ⊎ 𝟘)         -- the "End {ℓ}"
1255          -}
1256
1257          -- Pattern synonyms for more compact presentation
1258          pattern emptyM      = μ (inj₁ tt)                    -- : 𝕄
1259          pattern branchM l r = μ (inj₂ (inj₁ (l , r , tt)))   -- : 𝕄 → 𝕄 → 𝕄
              pattern absurdM a   = μ (inj₂ (inj₂ (inj₂ (inj₂ a)))) -- absurd values of 𝟘
1260
1261          data TreeSkeleton : Set where
1262            empty  : TreeSkeleton
1263            branch : TreeSkeleton → TreeSkeleton → TreeSkeleton
```

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

```
1265          𝕄→Tree : 𝕄 → TreeSkeleton
1266          𝕄→Tree emptyM = empty
1267          𝕄→Tree (branchM l r) = branch (𝕄→Tree l) (𝕄→Tree r)
1268          𝕄→Tree (absurdM (inj₁ ()))
1269          𝕄→Tree (absurdM (inj₂ ()))
1270
1271          𝕄←Tree : TreeSkeleton → 𝕄
1272          𝕄←Tree empty = emptyM
              𝕄←Tree (branch l r) = branchM (𝕄←Tree l) (𝕄←Tree r)
1273
1274
```

```
1275          M←Tree∘M→Tree : ∀ m → M←Tree (M→Tree m) ≡ m
1276          M←Tree∘M→Tree emptyM = refl
1277          M←Tree∘M→Tree (branchM l r) = cong₂ branchM (M←Tree∘M→Tree l) (M←Tree∘M→Tree r)
1278          M←Tree∘M→Tree (absurdM (inj₁ ()))
1279          M←Tree∘M→Tree (absurdM (inj₂ ()))

1280          M→Tree∘M←Tree : ∀ t → M→Tree (M←Tree t) ≡ t
1281          M→Tree∘M←Tree empty = refl
1282          M→Tree∘M←Tree (branch l r) = cong₂ branch (M→Tree∘M←Tree l) (M→Tree∘M←Tree r)
```

A.13 :kind

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

macro
  _:kind_ : Term → Term → Term → TC Unit.⊤
  _:kind_ t (con (quote 'record) _)    goal = normalise (t app (quoteTerm 0))
                        >>=ₘ λ t' → unify (waist-helper 0 t') goal
  _:kind_ t (con (quote 'typeclass) _) goal = normalise (t app (quoteTerm 1))
                        >>=ₘ λ t' → unify (waist-helper 1 t') goal
  _:kind_ t (con (quote 'data) _) goal = normalise (t app (quoteTerm 1))
                        >>=ₘ λ t' → normalise (waist-helper 1 t')
                        >>=ₘ λ t'' → unify goal (def (quote Fix) ((vArg (Σ→⊎₀ (sources₁ t''))) :: []))
  _:kind_ t _ goal = unify t goal
```

Informally, _:kind_ behaves as follows:

$$C \text{ :kind 'record} = C \text{ :waist } 0$$
$$C \text{ :kind 'typeclass} = C \text{ :waist } 1$$
$$C \text{ :kind 'data} = \text{termtype } (C \text{ :waist } 1)$$

A.14 termtype PointedSet ≅ 𝟙

```
-- termtype (PointedSet) ≅ ⊤ !
One  : Context (ℓsuc ℓ₀)
One       = do Carrier ← Set ℓ₀
               point   ← Carrier
               End {ℓ₀}

𝕆𝕟𝕖 : Set
𝕆𝕟𝕖 = termtype (One :waist 1)

view₁ : 𝕆𝕟𝕖 → 𝟙
view₁ emptyM = tt
```

A.15 The Termtype 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!

```
PointedOver₂ : Set → Context (ℓsuc ℓ₀)
PointedOver₂ Ξ    = do Carrier ← Set ℓ₀
                       relation ← (Ξ → Ξ → Carrier)
                       End {ℓ₀}

ℙ₂ : Set → Set
ℙ₂ X = termtype (PointedOver₂ X :waist 1)
```

```
1324          pattern _⇌_ x y = μ (inj₁ (x , y , tt))
1325
1326          view₂ : ∀ {X} → ℙ₂ X → X × X
1327          view₂ (x ⇌ y) = x , y
```

A.16   No 'constants', whence a type of inifinitely branching terms

```
1329          PointedOver₃  : Set → Context (ℓ₀)
1330          PointedOver₃ Ξ    = do relation ← (Ξ → Ξ → Ξ)
1331                                 End {ℓ₀}
1332
1333          ℙ₃ : Set
1334          ℙ₃ = termtype (λ X → PointedOver₃ X 0)
```

A.17   ℙ₂ again!

```
1337          PointedOver₄  : Context (ℓsuc ℓ₀)
1338          PointedOver₄        = do Ξ ← Set
1339                                   Carrier ← Set ℓ₀
1340                                   relation ← (Ξ → Ξ → Carrier)
1341                                   End {ℓ₀}
1342
1343          -- The current implementation of "termtype" only allows for one "Set" in the body.
1344          -- So we lift both out; thereby regaining ℙ₂!
1345
1346          ℙ₄ : Set → Set
1347          ℙ₄ X = termtype ((PointedOver₄ :waist 2) X)
1348
1349          pattern _⇌_ x y = μ (inj₁ (x , y , tt))
1350
1351          case₄ : ∀ {X} → ℙ₄ X → Set₁
1352          case₄ (x ⇌ y) = Set
1353
1354          -- Claim: Mention in paper.
1355          --
1356          --    P₁ : Set → Context = λ Ξ → do ⋯ End
1357          -- ≅  P₂ :waist 1
1358          -- where P₂ : Context = do Ξ ← Set; ⋯ End
```

A.18   ℙ₄ again – indexed unary algebras; i.e., "actions"

```
1357          PointedOver₈  : Context (ℓsuc ℓ₀)
1358          PointedOver₈        = do Index     ← Set
1359                                   Carrier   ← Set
1360                                   Operation ← (Index → Carrier → Carrier)
1361                                   End {ℓ₀}
1362
1363          ℙ₈ : Set → Set
1364          ℙ₈ X = termtype ((PointedOver₈ :waist 2) X)
1365
1366          pattern _·_ x y = μ (inj₁ (x , y , tt))
1367
1368          view₈ : ∀ {I} → ℙ₈ I → Set₁
1369          view₈ (i · e) = Set
```

**COMMENT Other experiments

```
1369          {- Yellow:
1370
1371          PointedOver₅  : Context (ℓsuc ℓ₀)
1372
```

```
PointedOver₅  = do One ← Set
                     Two ← Set
                     Three ← (One → Two → Set)
                     End {ℓ₀}

ℙ₅ : Set → Set₁
ℙ₅ X = termtype ((PointedOver₅ :waist 2) X)
-- Fix (λ Two → One × Two)

pattern _::₅_ x y = μ (inj₁ (x , y , tt))

case₅ : ∀ {X} → ℙ₅ X → Set₁
case₅ (x ::₅ xs) = Set

-}

--------------------------------------------------------------------------------

{-- Dependent sums

PointedOver₆  : Context ℓ₁
PointedOver₆ = do Sort ← Set
                     Carrier ← (Sort → Set)
                     End {ℓ₀}

ℙ₆ : Set₁
ℙ₆ = termtype ((PointedOver₆ :waist 1) )
-- Fix (λ X → X)

-}

--------------------------------------------------------------------------------

-- Distinuighed subset algebra

open import Data.Bool renaming (Bool to 𝔹)

{-
PointedOver₇  : Context (ℓsuc ℓ₀)
PointedOver₇      = do Index ← Set
                        Is   ← (Index → 𝔹)
                        End {ℓ₀}

-- The current implementation of "termtype" only allows for one "Set" in the body.
-- So we lift both out; thereby regaining ℙ₂!

ℙ₇ : Set → Set
ℙ₇ X = termtype (λ (_ : Set) → (PointedOver₇ :waist 1) X)
-- ℙ₁ X ≅ X

pattern _⇌_ x y = μ (inj₁ (x , y , tt))

case₇ : ∀ {X} → ℙ₇ X → Set
case₇ {X} (μ (inj₁ x)) = X

-}
```

```
1422            --------------------------------------------------------------------------------
1423
1424            {-
1425            PointedOver₉  : Context ℓ₁
               PointedOver₉      = do Carrier ← Set
1426                                   End {ℓ₀}
1427
1428            -- The current implementation of "termtype" only allows for one "Set" in the body.
1429            -- So we lift both out; thereby regaining ℙ₂!
1430
               ℙ₉ : Set
1431           ℙ₉ = termtype (λ (X : Set) → (PointedOver₉ :waist 1) X)
1432           -- ≅ 𝟘 ≅ Fix (λ X → 𝟘)
1433           -}
1434
1435   A.19  Fix Id
               PointedOver₁₀  : Context ℓ₁
1436           PointedOver₁₀      = do Carrier ← Set
1437                                   next    ← (Carrier → Carrier)
1438                                   End {ℓ₀}
1439
1440           -- The current implementation of "termtype" only allows for one "Set" in the body.
1441           -- So we lift both out; thereby regaining ℙ₂!
1442
               ℙ₁₀ : Set
1443           ℙ₁₀ = termtype (λ (X : Set) → (PointedOver₁₀ :waist 1) X)
1444           -- Fix (λ X → X), which does not exist.
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
```