Monadically Making Modules

—ICFP Deadline: March 3, 2020—

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 interface for a shared declaration interface, which is extensible in the language, we also find that common data structures correspond to simple theories.

ACM Reference Format:

. 2020. Monadically Making Modules —ICFP Deadline: March 3, 2020—. 1, 1 (March 2020), 14 pages.

https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We routinely write algebraic datatypes to provide a first-class syntax for record values. We work with semantic values, but need syntax to provide serialisation and introspection capabilities. A concept is thus rendered twice, one at the semantic level using records and again at the syntactic level using algebraic datatypes. Even worse, there is usually a need to expose fields of a record at the type level and so yet another variation of the same concept needs to be written. Our idea is to unify the two type declarations into one —using monadic do-notation and in-language meta-programming combinators to then extract possibly parameterised records and algebraic data types.

For example, there are two ways to implement the type of graphs in the dependently-typed language Agda: Having the vertices be a parameter or having them be a field of the record. Then there is also the syntax for graph vertex relationships.

```
record Graph₀ : Set₁ where
  constructor ⟨_,_⟩₀
  field
    Vertex : Set
```

Author's address:

Manuscript submitted to ACM

```
    Edges : Vertex → Vertex → Set

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

data Graph (Vertex : Set) : Set where
  ⟨_,_⟩ₛ : Vertex → Vertex → Graph Vertex
```

To illustrate the difference of the first two, consider the function `comap`, which relabels the vertices of a graph, using a function `f` to transform vertices:

```
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

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

In $\texttt{comap}_0$, the input graph `G` and the output graph `H` have their vertex sets constrained to match the type of the relabelling function `f`. Without the constraints, we could not even right the function for $\texttt{Graph}_0$. With such an importance, it is surprising to see that the occurrences of the constraint proofs are uninsightful `refl`-exivity proofs. In contrast, $\texttt{comap}_1$ does not carry any excesses baggage at the type level nor at the implementation level.

We will show an automatic technique for obtaining the above three definitions of graphs from a single declaration using similar notation. Our contributions are to show:

(1) Languages with sufficiently powerful type systems and meta-programming can conflate record and termtype declarations into one practical interface. 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).
(3) Programming with fixed-points of unary type constructors can be made as simple as programming with termtypes (Section 4).

- Astonishingly, we mechanically regain ubiquitous data structures such as `ℕ, Maybe, List` as the termtypes of simple pointed and monoidal theories.

As an application, in Section 5 we show that the resulting setup applies as a semantics of pre-processing tool that accomplishes the above tasks.

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
    States : Set
    start  : States
    next   : States → States

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

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

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**. 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 the *type constructor* `λ X : Set • Set`. We shall refer to the latter change as *reification* since the result is more concrete, it can be applied; it will be denoted by Π→λ.

Of-course, there is also the need for descriptions of values, which leads to the following termtypes. We shall refer to the shift from record types to algebraic data types as **the termtype problem**.

```
data DSTerms₀ : Set where
  start : DSTerms₀
  next  : DSTerms₀ → DSTerms₀

data DSTerms₁ (States : Set) : Set where
  start : States → DSTerms₁ States
  next  : DSTerms₁ States → DSTerms₁ States

data DSTerms₂ (States : Set) (start : States) : Set where
  next : DSTerms₂ States start → DSTerms₂ States start
```

Table 1. 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 • 1`	"bundled-up data"
Function Type	`Π S • Σ s : S • Σ n : S → S • 1`	"a type of functions"
Type constructor	`λ S • Σ s : S • Σ n : S → S • 1`	"a function on types"
Algebraic datatype	`data D : Set where s : D; n : D → D`	"a descriptive syntax"

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:

```
DynamicSystem : Context ℓ₁
DynamicSystem = do X ← Set
                   z ← X
                   s ← (X → X)
                   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 1.

Exposure	Elaboration
0	Σ X : Set • Σ z : X • Σ s : (X → X) • 1
1	Π X : Set • Σ z : X • Σ s : (X → X) • 1
2	Π X : Set • Π z : X • Σ s : (X → X) • 1
3	Π X : Set • Π z : X • Π s : (X → X) • 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 is a context -}
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S

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

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

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

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

```
‘ Set >>= λ X → ‘ X >>= λ z → ‘ (X → X) >>= End
{- or -}
do X ← ‘ Set
   z ← ‘ X
   s ← ‘ (X → X)
   End
```

With our goal of practicality in mind, we shall "build the lifting quote into the definition" of _>>=_:

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

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

```
𝒩₀ : DynamicSystem ∅  {- ≈ Σ X : Set  • Σ z : X  • Σ s : (X → X)  • 1 -}
𝒩₀ = ℕ , ∅ , suc , tt

𝒩₁ : DynamicSystem 1 {- ≈ Π X : Set  • Σ z : X  • Σ s : (X → X)  • 1 -}
𝒩₁ = λ X → ??? {- Impossible is X is empty! -}

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

It seems what we need is method, say Π→λ, that takes a Π-type and transforms it into a λ-expression. One could use a universe, an algebraic type of codes denoting types, to define Π→λ. However, one can no longer then easily use existing types since they are not formed from the universe's constructors, thereby resulting in duplication of existing types via the universe encoding. This is 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 Π→λ is thus a macro that acts on the syntactic term representations of types.

```
Π→λ (Π a : A • Ba) = (λ a : A • Ba)
{- One then extends this homomorphically over all possible term formers. -}
```

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

```
  Π→λ (Π→λ (DynamicSystem 2))
≡ Π→λ (Π→λ (Π X : Set • Π s : X  • Σ n : X → X  • 1))
≡ Π→λ (λ X : Set • Π s : X  • Σ n : X → X  • 1)
≡ λ X : Set • λ s : X  • Σ n : X → X  • 1
```

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

```
τ :waist n      = Π→λⁿ n (τ n)

Π→λⁿ 0         τ = τ
Π→λⁿ (n + 1) τ = Π→λⁿ n (Π→λ τ)
```

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
```

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 have two isomorphic copies of the integers, called `Sum` and `Prod`, whose reason for being is to distinguish two common monoids: The latter 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
              Id       ← Carrier
              _⊕_      ← (Carrier → Carrier → 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 ℓ₀ :waist 2) M ⊕` is the type of monoids over *particular* types `M` and *particular* operations ⊕. Of-course, this is orthogonal, since Haskell's use-case is for canonical typeclasses, which utilise unification on the carrier type `M` to find instance implementations.

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.

In particular, the description language D for dynamical systems, ref:contexts-table, declares concrete constructors for the fixpoint of `F`:

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

That is, D ≅ `Fix F` where:

```
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.

```
  DynamicSystem                            {- Definition -}
⟶ do S ← Set; s ← S; n ← (S → S); End  {- Wait 1      -}
⟶ λ S • Σ s : S • Σ n : S → S • 1        {- Replace products with sums -}
⟶ λ S •        S ⊎       (S → S) ⊎ 0     {- Isomorphisms; definition -}
⟶ F
```

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 nullary constructor `s : S` targets the type `S` and so its source is 1 —since we're introducing unit types, any existing unit types are dropped via 0.

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

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

Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (⇊ Ba)

{- Extend "sources, Σ→⊎" homomorphicly to other syntactic constructs -}

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

The hint "`Replace products with sums`" in the above calculation is realised formally as `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 `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)
```

Astonishingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if C `: Set → Context` ℓ_0 then C`' = λ X → termtype (`C` X :waist 1)` can be used to form 'free, lawless, C-instances'.

Table 2. Data strcutrues as free theories

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

The final correspondence 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 M and `TreeSkeleton`. Hint: Interactively case-split on values of M until the declared patterns appear, then replace 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 0

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

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 $P \cong$ `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
```

5 RELATED WORKS

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

The PackageFormer cite:DBLP:conf/gpce/Al-hassyCK19,alhassy$_{thesisproposal}$ 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 cite:10.5555/229872 to be invoked in the construction of complex grouping mechanisms.

Table 3. 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 eye to 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 functions:

Table 4. Contexts as a semantics for PackageFormer constructs

Syntax	Semantics
`PackageFormer`	`Context`
`:waist`	`:waist`
$\rightarrow\!\!\!\oplus\!\!\rightarrow$	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 is legitimate Agda syntax. Since 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 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 NEXT STEPS

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.

7 APPENDIX: WHAT ABOUT THE META-LANGUAGE'S PARAMETERS? MAYBEDELETE

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 termined with `End`.

```
PointedSet : Context ℓ₁
PointedSet = do Carrier ← Set
```

```
              point    ← Carrier
              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.

```
PointedPF : (Ξ : Set₁) → Context ℓ₁
PointedPF Ξ = do Carrier ← Set
                 point   ← Carrier
                 ‘ Ξ
```

Clearly `PointedPF 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.

```
-- Convenience names
PointedSetᵣ = PointedSet         :kind ‘record
PointedPFᵣ  = λ Ξ → PointedPF Ξ :kind ‘record

-- An extended record type: Two types with a point of each.
TwoPointedSets = PointedPFᵣ PointedSetᵣ

_ :   TwoPointedSets
    ≡ ( Σ Carrier₁ : Set • Σ point₁ : Carrier₁
      • Σ Carrier₂ : Set • Σ point₂ : Carrier₂ • 1)
_ = refl

-- Here's an instance
one : PointedSet :kind ‘record
one = 𝔹 , false , tt

-- Another; a pointed natural extended by a pointed bool,
-- with particular choices for both.
two : TwoPointedSets
two = ℕ , 0 , one
```

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

```
_PointedSets : ℕ → Set₁
zero  PointedSets = 1
suc n PointedSets = PointedPFᵣ (n PointedSets)
```

```
_ :   4 PointedSets
    ≡ (Σ Carrier₁ : Set • Σ point₁ : Carrier₁
      • Σ Carrier₂ : Set • Σ point₂ : Carrier₂
      • Σ Carrier₃ : Set • Σ point₃ : Carrier₃
      • Σ Carrier₄ : Set • Σ point₄ : Carrier₄ • 1)
_ = 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)
```

REFERENCES