Indexed Programming

Release 1.0

Pierre-Évariste Dagand

Feb 23, 2018

CONTENTS

Last week:

- monadic programming: "how to survive without (some) effects"
- took advantage of dependent types for proofs (tactics, anyone?)
- but wrote **simply**-typed programs (mostly)

Today, we shall:

- write dependently-typed program: types will dependent on values
- exploit inductive families to encode our invariants ("syntax")
- take advantage of this precision to host these domain-specific languages ("semantics")

The vision: The Proof Assistant as an Integrated Development Environment

Takeaways:

- you will be *able* to use inductive families to encode structural invariants
- you will be *able* to write dependently-typed programs over inductive families
- you will be *able* to construct denotational models in type theory
- you will be *familiar* with the Yoneda lemma
- you will be *familiar* with the notion of functor and presheaf
- you will be *familiar* with normalization-by-evaluation for the simply-typed calculus

CHAPTER

ONE

TYPING SPRINTF

Our introductory example is a Classic, introduced by Lennart Augustsson in his seminal paper on Cayenne. In plain ML (*ie.* without GADTs), the `sprintf` function cannot be given an ML type: the value of its arguments depending on the user-provided format.

```
sprintf "foo %d"    : ℕ → String
sprintf "bar %s"    : String → String
sprintf "baz %d %s" : ℕ → String → String
```

Formats are not random strings of characters:

- structure = syntax (*format*)
- from string to structure = parser

```
data format : Set where
  digit  : (k : format) → format
  string : (k : format) → format
  symb   : (c : Char)(k : format) → format
  end    : format

parse : List Char → format
parse ('%' :: 'd' :: cs ) = digit (parse cs)
parse ('%' :: 's' :: cs ) = string (parse cs)
parse ('%' :: c :: cs )   = symb c (parse cs)
parse ( c :: cs)          = symb c (parse cs)
parse []                  = end
```

We can *embed* the semantics of a format by describing its meaning within Agda itself:

```
⟦_⟧ : format → Set
⟦ digit k ⟧  = ℕ → ⟦ k ⟧
⟦ string k ⟧ = String → ⟦ k ⟧
⟦ symb c k ⟧ = ⟦ k ⟧
⟦ end ⟧      = String

⟦p_⟧ : String → Set
⟦p_⟧ = ⟦_⟧ ∘ parse ∘ toList
```

And we can easily realize this semantics:

```
eval : (fmt : format) → String → ⟦ fmt ⟧
eval (digit k) acc  = λ n → eval k (acc ++ showNat n)
eval (string k) acc = λ s → eval k (acc ++ s)
eval (symb c k) acc = eval k (acc ++ fromList (c :: []))
eval end acc        = acc
```

```
sprintf : (fmt : String) → ⟦p fmt ⟧
sprintf fmt = eval (parse (toList fmt)) ""
```

`sprintf` can thus be seen as an interpreter for a small language (whose AST is described by `format`) to the semantic domain described by `⟦_⟧`. And it works:

```
test : sprintf "test %%d & %%s: %d & %s" 2 "hello world!"
       ≡ "test %d & %s: S(S(0)) & hello world!"
test = refl
```

Exercise (difficulty: 1) Print `%d` using decimal numbers instead of Peano numbers

Exercise (difficulty: 3) Add support for the format `%.ns` where n is a (decimal) number representing the maximum size of the prefix of the string `s` to be printed. Careful, the formats `%.s` or `%.cs` are non-sensical: this should impact either parsing or interpretation.

CHAPTER

TWO

MOSTLY CORRECT-BY-CONSTRUCTION COMPILER

Using dependent types (in particular, inductive families), we can bake our invariants in the datatypes we manipulate and make sure that they are preserved as we process them. The advantage is twofold:

- build *initial* models of a domain of interest (syntax!)
- total denotational semantics in Agda itself (interpreter!)

We illustrate this approach with another Classic, a draft of James McKinna & Joel Wright entitled A type-correct, stack-safe, provably correct, expression compiler. As suggested by the title, we are going to implement a correct-by-construction compiler from a language of expressions to a stack machine.

2.1 Type-safe representation

Because Agda's type system is extremely rich, we can in fact *absorb* the type discipline of expressions in Agda. In programming terms, we define a datatype `exp` that represents only well-typed expressions:

```
data typ : Set where
   nat bool : typ

sem : typ → Set
sem nat  = ℕ
sem bool = Bool

data exp : typ → Set where
  val  : ∀ {T} → (v : sem T) → exp T
  plus : (e₁ e₂ : exp nat) → exp nat
  ifte : ∀ {T} → (c : exp bool)(e₁ e₂ : exp T) → exp T
```

We define the semantics of this language by interpretation within Agda:

```
eval : ∀ {T} → exp T → sem T
eval (val v)        = v
eval (plus e₁ e₂)   = eval e₁ + eval e₂
eval (ifte c e₁ e₂) = if eval c then eval e₁ else eval e₂
```

If we were pedantic, we would call this a *denotational* semantics.

Note that we crucially rely on the fact that `sem` computes at the type level to ensure that, for example, the `if_then_else_` is performed on a Boolean and not a natural number. This is called a *tagless* interpreter. In a non-dependent setting, values would have carried a tag (discriminating them based on their type) and the evaluator would have to deal with type errors dynamically:

```
module Tagged where

  data value : Set where
    isNat  : (n : ℕ) → value
    isBool : (b : Bool) → value

  data exp' : Set where
    val  : (v : value) → exp'
    plus : (e₁ e₂ : exp') → exp'
    ifte : (c e₁ e₂ : exp') → exp'

  eval' : exp' → Maybe value
  eval' (val v) = just v
  eval' (plus e₁ e₂)
    with eval' e₁ | eval' e₂
  ... | just (isNat n₁)
      | just (isNat n₂) = just (isNat (n₁ + n₂))
  ... | _ | _ = nothing
  eval' (ifte c e₁ e₂)
    with eval' c | eval' e₁ | eval' e₂
  ... | just (isBool b) | v₁ | v₂ = if b then v₁ else v₂
  ... | _ | _ | _ = nothing
```

Exercise (difficulty: 1) The above implementation is needlessly verbose, use the Maybe monad to abstract away error handling.

The moral of this implementation is that we failed to encode our invariant in the datatype `exp'` and paid the price in the implementation of `eval'`.

2.2 Stack machine

Our stack machine will interpret a fixed set of opcodes, transforming input stack to output stack. A stack will contain values, ie. Booleans or integers. We can therefore describe well-typed stacks by identifying the type of each elements:

```
stack-typ = List typ

data stack : stack-typ → Set where
  ϵ   : stack []
  _•_ : ∀ {T σ} → sem T → stack σ → stack (T ∷ σ)
```

In particular, a non-empty stack allows us to peek at the top element and to take its tail:

```
top : ∀ {T σ} → stack (T ∷ σ) → sem T
top (t • _) = t

tail : ∀ {T σ} → stack (T ∷ σ) → stack σ
tail (_ • s) = s
```

Using an inductive family, we can once again garantee that instructions are only applied onto well-formed and well-typed stacks:

```
data code : stack-typ → stack-typ → Set where
  skip : ∀ {σ} → code σ σ
  _#_  : ∀ {σ₁ σ₂ σ₃} → (c₁ : code σ₁ σ₂)(c₂ : code σ₂ σ₃) → code σ₁ σ₃
  PUSH : ∀ {T σ} → (v : sem T) → code σ (T ∷ σ)
```

```
  ADD  : ∀ {σ} → code (nat :: nat :: σ) (nat :: σ)
  IFTE : ∀ {σ₁ σ₂} → (c₁ c₂ : code σ₁ σ₂) → code (bool :: σ₁) σ₂
```

As a result, we can implement a (total) interpreter for our stack machine:

```
exec : ∀ {σ-i σ-f} → code σ-i σ-f → stack σ-i → stack σ-f
exec skip s                  = s
exec (c₁ # c₂) s             = exec c₂ (exec c₁ s)
exec (PUSH v) s              = v • s
exec ADD (x₁ • x₂ • s)       = x₁ + x₂ • s
exec (IFTE c₁ c₂) (true • s)  = exec c₁ s
exec (IFTE c₁ c₂) (false • s) = exec c₂ s
```

Exercise (difficulty: 1) Implement a simply-typed version of `code` and update `exec` to work (partially) from list of tagged values to list of tagged values.

2.3 Compilation

The compiler from expressions to stack machine code is then straightforward, the types making sure that we cannot generate non-sensical opcodes:

```
compile : ∀ {T σ} → exp T → code σ (T :: σ)
compile (val v)        = PUSH v
compile (plus e₁ e₂)   = compile e₂ # compile e₁ # ADD
compile (ifte c e₁ e₂) = compile c # IFTE (compile e₁) (compile e₂)
```

Exercise (difficulty: 1) Implement the (same) compiler on the simply-typed representation of expressions `exp'`.

Note that this does not guarantee that we preserve the semantics!

Exercise (difficulty: 4) We could address that remark by indexing expressions (`exp`) not only by their type but also by their denotation (a natural number):

```
expSem : (T : typ) → ⟦ T ⟧ → Set
```

Similarly, the stack machine opcodes could be indexed by their denotation (a stack):

```
codeSem : (σ : stack-typ) → stack σ → Set
```

As a result, a type-safe `compile` function from `expSem` to `codeSem` could ensure semantics-preservation by construction. Implement these source and target languages and the correct-by-construction compiler.

2.4 Correctness

The correctness proof amounts to showing that the interpreter for expressions agrees with the result of executing the stack machine. Having baked the typing discipline in our input expressions and output machine codes, we can focus on proving only the meaningful cases:

```
correctness : forall {T σ} → (e : exp T)(s : stack σ) → exec (compile e) s ≡ eval e • s
correctness (val v) s = refl
correctness (plus e₁ e₂) s
  rewrite correctness e₂ s
```

```
          | correctness e₁ (eval e₂ • s) = refl
correctness (ifte c e₁ e₂) s
  rewrite correctness c s
  with eval c
... | true rewrite correctness e₁ s = refl
... | false rewrite correctness e₂ s = refl
```

Exercise (difficulty: 2) Prove the same theorem one the simply-typed implementations. You may prefer to work in Coq, so as to take advantage of tactics to automate the tedium.

This exercise has its roots in the very origin of most programming and reasoning techniques we take for granted today:

- the role of initiality in formal reasoning
- the importance of equational reasoning for proving program correctness

These ideas were, for examples, in their inception at the first edition of POPL with Advice on structuring compilers and proving them correct (1973), which was further refined by More on advice on structuring compilers and proving them correct, (1980). This reflects the influence it had on a generation of computer scientists interested in language design on one hand (they gave us algebraic datatypes) and verified compilation on the other hand (they gave us denotational models). I learned these ideas from Conor McBride, who had learned them from James McKinna.

CHAPTER

THREE

COMPUTING NORMAL FORMS OF λ-TERMS

In Lecture 1, we have seen that, by finding a suitable semantics domain, we could auto-magically compute normal forms for monadic programs. Could we do the same for the whole (effect-free) λ-calculus?

3.1 Types and terms

We consider the simply-typed λ-calculus, whose grammar of types and contexts is as follows:

```
data type : Set where
  unit    : type
  _⇒_ _*_ : (S T : type) → type

data context : Set where
  ϵ   : context
  _▹_ : (Γ : context)(T : type) → context
```

Thanks to inductive families, we can represent *exactly* the well-scoped and well-typed λ-terms:

```
data _∈_ (T : type) : context → Set where
  here  : ∀ {Γ} → T ∈ Γ ▹ T
  there : ∀{Γ T'} → (h : T ∈ Γ) → T ∈ Γ ▹ T'

data _⊢_ (Γ : context) : type → Set where
  lam : ∀{S T} →

      (b : Γ ▹ S ⊢ T) →
      ---------------
      Γ ⊢ S ⇒ T

  var : ∀{T} →

      (v : T ∈ Γ) →
      -----------
      Γ ⊢ T

  _!_ : ∀{S T} →

      (f : Γ ⊢ S ⇒ T)(s : Γ ⊢ S) →
      --------------------------
      Γ ⊢ T

  tt :
```

```
    --------
    Γ ⊢ unit

  pair : ∀{A B} →

    (a : Γ ⊢ A)(b : Γ ⊢ B) →
    ----------------------
    Γ ⊢ A * B

  fst : ∀{A B} →

    Γ ⊢ A * B →
    ---------
    Γ ⊢ A

  snd : ∀{A B} →

    Γ ⊢ A * B →
    ---------
    Γ ⊢ B
```

This representation of λ-terms is folklore amongst programmers of the dependent kind. A comprehensive discussion of its pros and cons can be found in the pedagogical Strongly Typed Term Representations in Coq.

3.2 Interlude: substitution, structurally

Substitution for de Bruijn λ-terms is usually (offhandedly) specified in the following manner:

```
n [σ]     = σ(n)
(M N)[σ] = M[σ] N[σ]
(λ M)[σ] = λ (M[0 · (σ ∘ λ n. suc n)])

σ ∘ ρ     = λ n. (σ n)[ρ]
```

However, this definition contains a fair amount of mutual recursion, whose validity is not obvious and will be a hard sell to a termination checker. Let us exhibit this structure and, at the same time, exercise ourselves in the art of unearthing initial models.

Exercise (difficulty: 2) In Agda, the type of finite sets of cardinality `n` is defined by an inductive family:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

We are interested in **monotone** functions from `Fin n` to `Fin m`. We could obviously formalize this class of functions as "any function from `Fin n` to `Fin m` as long as it is monotone" however a more *intentional* characterization can be given by means of an inductive family:

```
data _≥_ : (m : ℕ)(n : ℕ) → Set where
  -- COMPLETE
```

Intuitively, this datatype provides a grammar of monotone functions, which we can then interpret back into actual (monotone) functions:

```
⟦_⟧ : ∀ {m n} → m ≥ n → Fin n → Fin m
⟦ wk ⟧ k = {!!}

lemma-valid : ∀{m n k l} → (wk : m ≥ n) → k ≤ l → ⟦ wk ⟧ k ≤ ⟦ wk ⟧ l
lemma-valid = {!!}
```

We can adapt this intentional characterization of monotone functions to typed embeddings:

```
data _≥_ : context → context → Set where
  id    : ∀ {Γ} → Γ ≥ Γ
  weak1 : ∀ {Γ Δ A} → (wk : Δ ≥ Γ) → Δ ▹ A ≥ Γ
  weak2 : ∀ {Γ Δ A} → (wk : Δ ≥ Γ) → Δ ▹ A ≥ Γ ▹ A

shift : ∀ {Γ Δ T} → Γ ≥ Δ → T ∈ Δ → T ∈ Γ
shift id v                = v
shift (weak1 wk) v        = there (shift wk v)
shift (weak2 wk) here     = here
shift (weak2 wk) (there v) = there (shift wk v)

rename : ∀ {Γ Δ T} → Γ ≥ Δ → Δ ⊢ T → Γ ⊢ T
rename wk (lam t)    = lam (rename (weak2 wk) t)
rename wk (var v)    = var (shift wk v)
rename wk (f ! s)    = rename wk f ! rename wk s
rename wk tt         = tt
rename wk (pair a b) = pair (rename wk a) (rename wk b)
rename wk (fst p)    = fst (rename wk p)
rename wk (snd p)    = snd (rename wk p)

sub : ∀ {Γ Δ T} → Γ ⊢ T → (∀ {S} → S ∈ Γ →  Δ ⊢ S) → Δ ⊢ T
sub (lam t) ρ    = lam (sub t (λ { here      → var here ;
                                   (there v) → rename (weak1 id) (ρ v) }))
sub (var v) ρ    = ρ v
sub (f ! s) ρ    = sub f ρ ! sub s ρ
sub tt ρ         = tt
sub (pair a b) ρ = pair (sub a ρ) (sub b ρ)
sub (fst p) ρ    = fst (sub p ρ)
sub (snd p) ρ    = snd (sub p ρ)

sub1 : ∀ {Γ S T} → Γ ▹ S ⊢ T → Γ ⊢ S → Γ ⊢ T
sub1 t s = sub t (λ { here → s ; (there v) → var v })
```

A formal treatment of this construction can be found in Formalized metatheory with terms represented by an indexed family of types, for example.

Exercise (difficulty: 2) Weakenings interpret to renaming functions and functions do compose so we are naturally driven to implement composition directly on renamings:

```
_∘wk_ : ∀ {Δ ∇ Γ} → Δ ≥ ∇ → Γ ≥ Δ → Γ ≥ ∇
_∘wk_ = {!!}
```

And we must make sure, that this notion of composition is the *right* one:

```
lemma-right-unit : ∀ {Γ Δ} → (wk : Γ ≥ Δ) → wk ∘wk id ≡ wk
lemma-right-unit = {!!}

lemma-left-unit : ∀ {Γ Δ} → (wk : Γ ≥ Δ) → id ∘wk wk ≡ wk
lemma-left-unit = {!!}
```

```
lemma-assoc : ∀ {Γ Δ ∇ Ω} → (wk₃ : Γ ⊇ Δ)(wk₂ : Δ ⊇ ∇)(wk₁ : ∇ ⊇ Ω) →
  (wk₁ ∘wk wk₂) ∘wk wk₃ ≡ wk₁ ∘wk (wk₂ ∘wk wk₃)
lemma-assoc = {!!}
```

3.3 Normal forms

We can represent the equation theory as an inductive family:

```
data _⊢_∋_~βη_ : (Γ : context)(T : type) → Γ ⊢ T → Γ ⊢ T → Set where
  rule-β : ∀{Γ S T}{b : Γ ▹ S ⊢ T}{s : Γ ⊢ S} →

    ------------------------------------
    Γ ⊢ T ∋ (lam b) ! s ~βη sub1 b s

  rule-η-fun : ∀{Γ S T}{f : Γ ⊢ S ⇒ T} →

    ----------------------------------------------------
    Γ ⊢ S ⇒ T ∋ f ~βη lam (rename (weak1 id) f ! var here)

  rule-η-pair : ∀{Γ A B}{p : Γ ⊢ A * B} →

    ----------------------------------------------------
    Γ ⊢ A * B ∋ p ~βη pair (fst p) (snd p)


data _⊢_∋_~βη_  : (Γ : context)(T : type) → Γ ⊢ T → Γ ⊢ T → Set where
  inc : ∀ {Γ T t₁ t₂} →

    Γ ⊢ T ∋ t₁ ~βη t₂ →
    -----------------
    Γ ⊢ T ∋ t₁ ~βη t₂


  reflexivity : ∀{Γ T t} →

    -----------
    Γ ⊢ T ∋ t ~βη t

  symmetry : ∀{Γ T t t'} →

    Γ ⊢ T ∋ t ~βη t' →
    ------------
    Γ ⊢ T ∋ t' ~βη t

  transitivity : ∀{Γ T t t' t''} →

    Γ ⊢ T ∋ t ~βη t' →
    Γ ⊢ T ∋ t' ~βη t'' →
    --------------
    Γ ⊢ T ∋ t ~βη t''

  struct-lam : ∀{Γ S T b b'} →

    Γ ▹ S ⊢ T ∋ b ~βη b' →
    ----------------
```

```
    Γ ⊢ S ⇒ T ∋ lam b ~βη lam b'

  struct-! : ∀{Γ S T f f' s s'} →

    Γ ⊢ S ⇒ T ∋ f ~βη f' →
    Γ ⊢ S ∋ s ~βη s' →
    -----------------
    Γ ⊢ T ∋ f ! s ~βη f' ! s'

  struct-pair : ∀{Γ A B a a' b b'} →

    Γ ⊢ A ∋ a ~βη a' →
    Γ ⊢ B ∋ b ~βη b' →
    ----------------
    Γ ⊢ A * B ∋ pair a b ~βη pair a' b'

  struct-fst : ∀{Γ A B p p'} →

    Γ ⊢ A * B ∋ p ~βη p' →
    ------------------------
    Γ ⊢ A ∋ fst p ~βη fst p'

  struct-snd : ∀{Γ A B p p'} →

    Γ ⊢ A * B ∋ p ~βη p' →
    ------------------------
    Γ ⊢ B ∋ snd p ~βη snd p'
```

Compute η-long β-normal forms for the simply typed λ-calculus:

- define a representation of terms (`term`)
- interpret types and contexts in this syntactic model (`⟦_⟧` and `⟦_⟧context`)
- interpret terms in this syntactic model (`eval`)

```
data term : Set where
   lam  : (v : String)(b : term) → term
   var  : (v : String) → term
   _!_  : (f : term)(s : term) → term
   tt   : term
   pair : (x y : term) → term
   fst  : (p : term) → term
   snd  : (p : term) → term

⟦_⟧Type : type → Set
⟦ unit ⟧Type  = term
⟦ S ⇒ T ⟧Type = ⟦ S ⟧Type → ⟦ T ⟧Type
⟦ S * T ⟧Type = ⟦ S ⟧Type × ⟦ T ⟧Type

⟦_⟧context : context → Set
⟦ ∈ ⟧context     = ⊤
⟦ Γ ▹ T ⟧context = ⟦ Γ ⟧context × ⟦ T ⟧Type

_▯_ : context → type → Set
Γ ▯ T = ⟦ Γ ⟧context → ⟦ T ⟧Type

lookup : ∀{Γ T} → T ∈ Γ → Γ ▯ T
lookup here (_ , x)      = x
```

```
lookup (there h) (γ , _) = lookup h γ

eval : ∀{Γ T} → Γ ⊢ T → Γ ▯ T
eval (var v) ρ     = lookup v ρ
eval (f ! s) ρ     = eval f ρ (eval s ρ)
eval (lam b) ρ     = λ s → eval b (ρ , s)
eval (pair a b) ρ = eval a ρ , eval b ρ
eval (fst p) ρ     = proj₁ (eval p ρ)
eval (snd p) ρ     = proj₂ (eval p ρ)
eval tt ρ          = tt
```

This is an old technique, introduced by Per Martin-Löf in About Models for Intuitionistic Type Theories and the Notion of Definitional Equality, applied by Coquand & Dybjer to the simply-typed λ-calculus in Intuitionistic Model Constructions and Normalization Proofs.

Let us, for simplicity, assume that we have access to fresh name generator, `gensym`:

```
postulate gensym : ⊤ → String
```

This would be the case if we were to write this program in OCaml, for instance.

We could then back-translate the objects in the model (`⟦_⟧Type`) back to raw terms (through `reify`). However, to do so, one needs to inject variables *in η-long normal form* into the model: this is the role of `reflect`:

```
reify : ∀{T} → ⟦ T ⟧Type → term
reflect : (T : type) → term → ⟦ T ⟧Type

reify {unit} nf       = nf
reify {A * B} (x , y) = pair (reify x) (reify y)
reify {S ⇒ T} f       = let s = gensym tt in
                          lam s (reify (f (reflect S (var s))))

reflect unit nf     = nf
reflect (A * B) nf  = reflect A (fst nf) , reflect B (snd nf)
reflect (S ⇒ T) neu = λ s → reflect T (neu ! reify s)
```

Given a λ-term, we can thus compute its normal form:

```
norm :  ∀{Γ T} → Γ ⊢ T → term
norm {Γ} Δ = reify (eval Δ (idC Γ))
  where idC : ∀ Γ → ⟦ Γ ⟧context
        idC ϵ       = tt
        idC (Γ ▹ T) = idC Γ , reflect T (var (gensym tt))
```

Just like in the previous lecture (and assuming that we have proved the soundness of this procedure with respect to the equational theory `_⊢_∋_~`$\beta\eta$`_`), we can use it to check whether any two terms belong to the same congruence class by comparing their normal forms:

```
term₁ : ϵ ⊢ (unit ⇒ unit) ⇒ unit ⇒ unit
term₁ =
  -- λ s. λ z. s (s z)
  lam (lam (var (there here) ! (var (there here) ! var here)))

term₂ : ϵ ⊢ (unit ⇒ unit) ⇒ unit ⇒ unit
term₂ =
  -- λ s. (λ r λ z. r (s z)) (λ x. s x)
  lam (lam (lam (var (there here) ! (var (there (there here)) ! var here))) ! lam (var (there here) ! var␣
↪here))
```

```
test-nbe : norm term₁ ≡ norm term₂
test-nbe = refl
```

For instance, thanks to a suitable model construction, we have surjective pairing:

```
term₃ : ϵ ⊢ unit * unit ⇒ unit * unit
term₃ =
  -- λ p. p
  lam (var here)

term₄ : ϵ ⊢ unit * unit ⇒ unit * unit
term₄ =
  -- λ p. (fst p, snd p)
  lam (pair (fst (var here)) (snd (var here)))

test-nbe₂ : norm term₃ ≡ norm term₄
test-nbe₂ = refl
```

Exercise (difficulty: 4) Modify the model so as to remove surjective pairing (`rule-η-pair` would not be valid) while retaining the usual η-rule for functions (`rule-η-fun`). Hint: we have used the *negative* presentation of products which naturally leads to a model enabling η for pair. Using the *positive* presentation would naturally lead to one in which surjective pairing is not valid.

However, this implementation is a bit of wishful thinking: we do not have a `gensym`! So the following is also true, for the bad reason that `gensym` is not actually producing unique names but always the *same* name (itself):

```
term₅ : ϵ ⊢ unit ⇒ unit ⇒ unit
term₅ =
  -- λ z₁ z₂. z₁
  lam (lam (var (there here)))

term₆ : ϵ ⊢ unit ⇒ unit ⇒ unit
term₆ =
  -- λ z₁ z₂. z₂
  lam (lam (var here))

test-nbe₃ : norm term₅ ≡ norm term₆
test-nbe₃ = refl -- BUG!
```

This might not deter the brave monadic programmer: we can emulate `gensym` using a reenactment of the state monad:

```
Fresh : Set → Set
Fresh A = ℕ → A × ℕ

gensym : ⊤ → Fresh String
gensym tt = λ n → showNat n , 1 + n

return : ∀ {A} → A → Fresh A
return a = λ n → (a , n)

_>>=_ : ∀ {A B} → Fresh A → (A → Fresh B) → Fresh B
m >>= k = λ n → let (a , n') = m n in k a n'

run : ∀ {A} → Fresh A → A
run f = proj₁ (f 0)
```

We then simply translate the previous code to a monadic style, a computer could do it automatically:

```
reify : ∀{T} → ⟦ T ⟧Type → Fresh term
reflect : (T : type) → term → Fresh ⟦ T ⟧Type

reify {unit} nf       = return nf
reify {A * B} (a , b) = reify a >>= λ a →
                        reify b >>= λ b →
                        return (pair a b)
reify {S ⇒ T} f       = gensym tt >>= λ s →
                        reflect S (var s) >>= λ t →
                        reify (f t) >>= λ b →
                        return (lam s b)

reflect unit nf     = return nf
reflect (A * B) nf  = reflect A (fst nf) >>= λ a →
                      reflect B (snd nf) >>= λ b →
                      return (a , b)
reflect (S ⇒ T) neu = return (λ s → {!!})
-- XXX: cannot conclude with `reflect T (neu ! reify s)`
```

Excepted that, try as we might, we cannot reflect a function.

Exercise (difficulty: 1) Try (very hard) at home. Come up with a simple explanation justifying why it is impossible.

Exercise (difficulty: 3) Inspired by this failed attempt, modify the syntactic model with the smallest possible change so as to be able to implement `reify`, `reflect` and obtain a valid normalisaton function. Hint: a solution is presented in Normalization and Partial Evaluation.

3.4 The Rising Sea

Rather than hack our model, I propose to gear up and let the sea rise because "when the time is ripe, hand pressure is enough". Another argument against incrementally improving our model is its fragility: whilst our source language is well structured (well-scoped, well-typed λ-terms), our target language (raw λ-terms) is completely destructured, guaranteeing neither that we actually produce normal forms, nor that it is well-typed not even proper scoping.

To remedy this, let us

- precisely describe η-long β-normal forms
- check that they embed back into well-typed, well-scoped terms

```
data _⊢Nf_ (Γ : context) : type → Set
data _⊢Ne_ (Γ : context) : type → Set

data _⊢Nf_ (Γ : context) where
     lam    : ∀ {S T} → (b : Γ ▹ S ⊢Nf T) → Γ ⊢Nf S ⇒ T
     pair   : ∀ {A B} → Γ ⊢Nf A → Γ ⊢Nf B → Γ ⊢Nf A * B
     tt     : Γ ⊢Nf unit
     ground : (grnd : Γ ⊢Ne unit) → Γ ⊢Nf unit

data _⊢Ne_ (Γ : context) where
   var : ∀{T} → (v : T ∈ Γ) → Γ ⊢Ne T
   _!_ : ∀{S T} → (f : Γ ⊢Ne S ⇒ T)(s : Γ ⊢Nf S) → Γ ⊢Ne T
```

```
   fst : ∀ {A B} → (p : Γ ⊢Ne A * B) → Γ ⊢Ne A
   snd : ∀ {A B} → (p : Γ ⊢Ne A * B) → Γ ⊢Ne B

⌊_⌋Ne : ∀{Γ T} → Γ ⊢Ne T → Γ ⊢ T
⌊_⌋Nf : ∀{Γ T} → Γ ⊢Nf T → Γ ⊢ T

⌊ lam b ⌋Nf       = lam ⌊ b ⌋Nf
⌊ ground grnd ⌋Nf = ⌊ grnd ⌋Ne
⌊ pair a b ⌋Nf    = pair ⌊ a ⌋Nf ⌊ b ⌋Nf
⌊ tt ⌋Nf          = tt

⌊ var v ⌋Ne       = var v
⌊ f ! s ⌋Ne       = ⌊ f ⌋Ne ! ⌊ s ⌋Nf
⌊ fst p ⌋Ne       = fst ⌊ p ⌋Ne
⌊ snd p ⌋Ne       = snd ⌊ p ⌋Ne
```

We are going to construct a context-and-type-indexed model

```
[_]□_ : context → type → Set
```

(reading `[ Γ ]□ T` as "an interpretation of `T` in context Γ) so as to ensure that the normal forms we produce by reification are well-typed and well-scoped (and, conversely, to ensure that the neutral terms we reflect are necessarily well-typed and well-scoped). The types of `reify` and `reflect` thus become:

```
reify   : ∀ {Γ T} → [ Γ ]□ T  → Γ ⊢Nf T
reflect : ∀ {Γ} → (T : type) → Γ ⊢Ne T → [ Γ ]□ T
```

However, we expect some head-scratching when implementing `reify` on functions: this is precisely were we needed the `gensym` earlier. We can safely assume that function application is admissible in our model, ie. we have an object

```
app : ∀ {Γ S T} → [ Γ ]□ S ⇒ T → [ Γ ]□ S → [ Γ ]□ T
```

Similarly, using `reflect`, we can easily lift the judgment `var here : Γ ▷ S ⊢ S` into the model:

```
reflect S (var here) : [ Γ ▷ S ]□ S
```

It is therefore tempting to implement the function case of `reify` as follows:

```
reify {S ⇒ T} f = lam (reify (app f (reflect S (var here))))
```

However, `f` has type `[ Γ ]□ S ⇒ T` and we are working under a lambda, in the context `Γ ▷ S`. We need a weakening operator (denoted `ren`) in the model! Then we could just write:

```
reify {S ⇒ T} f = lam (reify (app (ren (weak1 id) f) (reflect S (var here))))
```

Remark: We do not make the mistake of considering a (simpler) weakening from Γ to `Γ ▷ S`. As usual (eg. `rename` function earlier), such a specification would not be sufficiently general and we would be stuck when trying to go through another binder. Even though we only use it with `weak1 id`, the weakening operator must therefore be defined over any weakening.

Translating these intuitions into a formal definition, this means that our semantics objects are context-indexed families that come equipped with renaming operation:

```
record Sem : Set₁ where
  field
```

```
    _⊢ : context → Set
    ren : ∀ {Γ Δ} → Γ ⊇ Δ → Δ ⊢ → Γ ⊢
```

An implication in `Sem` is a family of implications for each context:

```
_→_ : (P Q : Sem) → Set
P → Q = ∀ {Γ} → Γ ⊢P → Γ ⊢Q
  where open Sem P renaming (_⊢ to _⊢P)
        open Sem Q renaming (_⊢ to _⊢Q)
```

We easily check that normal forms and neutral terms implement this interface:

```
rename-Nf : ∀{Γ Δ T} → Γ ⊇ Δ → Δ ⊢Nf T → Γ ⊢Nf T
rename-Ne : ∀{Γ Δ T} → Γ ⊇ Δ → Δ ⊢Ne T → Γ ⊢Ne T

rename-Nf wk (lam b)       = lam (rename-Nf (weak2 wk) b)
rename-Nf wk (ground grnd) = ground (rename-Ne wk grnd)
rename-Nf wk (pair a b)    = pair (rename-Nf wk a) (rename-Nf wk b)
rename-Nf wk tt            = tt

rename-Ne wk (var v)       = var (shift wk v)
rename-Ne wk (f ! s)       = (rename-Ne wk f) ! (rename-Nf wk s)
rename-Ne wk (fst p)       = fst (rename-Ne wk p)
rename-Ne wk (snd p)       = snd (rename-Ne wk p)

Nf̂ : type → Sem
Nf̂ T = record { _⊢ = λ Γ → Γ ⊢Nf T
               ; ren = rename-Nf }

Nê : type → Sem
Nê T = record { _⊢ = λ Γ → Γ ⊢Ne T
              ; ren = rename-Ne }
```

Following our earlier model, we will interpret the `unit` type as the normal forms of type unit:

```
⟦unit⟧ : Sem
⟦unit⟧ =  Nf̂ unit

⟦tt⟧ : ∀ {P} → P → ⟦unit⟧
⟦tt⟧ ρ = tt
```

Similarly, we will interpret the `_*_` type as a product in `Sem`, defined in a pointwise manner:

```
_⟦×⟧_ : Sem → Sem → Sem
P ⟦×⟧ Q = record { _⊢ = λ Γ → Γ ⊢P × Γ ⊢Q
                ; ren = λ { wk (x , y) → ( ren-P wk x , ren-Q wk y ) } }
  where open Sem P renaming (_⊢ to _⊢P ; ren to ren-P)
        open Sem Q renaming (_⊢ to _⊢Q ; ren to ren-Q)

⟦pair⟧ : ∀ {P Q R} → P → Q → P → R → P → Q ⟦×⟧ R
⟦pair⟧ a b ρ = a ρ , b ρ

⟦fst⟧ : ∀ {P Q R} → P → Q ⟦×⟧ R → P → Q
⟦fst⟧ p ρ = proj₁ (p ρ)

⟦snd⟧ : ∀ {P Q R} → P → Q ⟦×⟧ R → P → R
⟦snd⟧ p ρ = proj₂ (p ρ)
```

We may be tempted to define the exponential in a pointwise manner too:

```
_⟦⇒⟧_ : Sem → Sem → Sem
P ⟦⇒⟧ Q = record { _⊢ = λ Γ → Γ ⊢P → Γ ⊢Q
                ; ren = ?! }
  where open Sem P renaming (_⊢ to _⊢P)
        open Sem Q renaming (_⊢ to _⊢Q)
```

However, we are bitten by the contra-variance of the domain: there is no way to implement `ren` with such a definition.

3.5 Interlude: Yoneda lemma

Let `_⊢T : context → Set` be a semantics objects together with its weakening operator `ren-T : Γ ⊇ Δ → Δ ⊢T → Γ ⊢T`. By mere application of `ren-T`, we can implement:

```
ψ : Γ ⊢T → (∀ {Δ} → Δ ⊇ Γ → Δ ⊢T)
ψ t wk = ren-T wk t
```

were the `∀ {Δ} →` quantifier of the codomain type must be understood in polymorphic sense. Surprisingly (perhaps), we can go from the polymorphic function back to a single element, by providing the `id` continuation:

```
ϕ : (∀ {Δ} → Δ ⊇ Γ → Δ ⊢T) → Γ ⊢T
ϕ k = k id
```

One could then prove that this establishes an isomorphism, for all Γ:

```
postulate
  ψ∘ϕ≡id : ψ ∘ ϕ ≡ λ k → k
  ϕ∘ψ≡id : ϕ ∘ ψ ≡ λ t → t
```

Exercise (difficulty: 4) To prove this, we need to structural results on Sem, which we have eluded for now (because they are not necessary for programming). Typically, we would expect that `ren` on the identity weakening `id` behaves like an identity, etc. Complete the previous definitions so as to provide these structural lemmas and prove the isomorphism.

A slightly more abstract way of presenting this isomorphism consists in noticing that any downward-closed set of context forms a valid semantics objects. ϕ and ψ can thus be read as establishing an isomorphism between the object `Γ ⊢T` and the morphisms in `⊇[ Γ ] → T`:

```
⊇[_] : context → Sem
⊇[ Γ ] = record { _⊢ = λ Δ → Δ ⊇ Γ
                ; ren = λ wk₁ wk₂ → wk₂ ∘wk wk₁ }

ψ' : Γ ⊢T → ⊇[ Γ ] → T
ψ' t wk = ren-T wk t

ϕ' : ⊇[ Γ ] → T → Γ ⊢T
ϕ' k = k id
```

Being isomorphic to `_ ⊢T`, we expect the type `λ Γ → ∀ {Δ} → Δ ⊇ Γ → Δ ⊢T` to be a valid semantic object. This is indeed the case, where renaming merely lifts composition of weakening:

```
Y : Sem
Y = record { _⊢ = λ Γ → ∀ {Δ} → Δ ⊇ Γ → Δ ⊢T
           ; ren = λ wk₁ k wk₂ → k (wk₁ ∘wk wk₂) }
```

Note that `Y` does not depend on `ren-T`: it is actually baked in the very definition of `_⊢`!

3.6 Back to the Sea

Let us assume that the exponential `P ⟦⇒⟧ Q : Sem` exists. This means, in particular, that it satisfies the following isomorphism for all `R : Sem`:

```
R → P ⟦⇒⟧ Q ≡ R ⟦×⟧ P → Q
```

We denote `_⊢P⟦⇒⟧Q` its action on contexts. Let `Γ : context`. We have the following isomorphisms:

```
Γ ⊢P⟦⇒⟧Q ≡ ∀ {Δ} → Δ ⊇ Γ → Δ ⊢P⟦⇒⟧Q              -- by ψ
       ≡ ⊇[ Γ ] → P⟦⇒⟧Q                      -- by the alternative definition ψ'
       ≡ ⊇[ Γ ] ⟦×⟧ P → Q                    -- by definition of an exponential
       ≡ ∀ {Δ} → Δ ⊇ Γ → Δ ⊢P → Δ ⊢Q         -- by unfolding definition of ⟦×⟧, → and currying
```

As in the definition of `Y`, it is easy to see that this last member can easily be equipped with a renaming: we therefore take it as the **definition** of the exponential:

```
_⟦⇒⟧_ : Sem → Sem → Sem
P ⟦⇒⟧ Q = record { _⊢ = λ Γ → ∀ {Δ} → Δ ⊇ Γ → Δ ⊢P → Δ ⊢Q
                ; ren = λ wk₁ k wk₂ → k (wk₁ ∘wk wk₂) }
  where open Sem P renaming (_⊢ to _⊢P)
        open Sem Q renaming (_⊢ to _⊢Q)

⟦lam⟧ : ∀ {P Q R} → P ⟦×⟧ Q → R → P → Q ⟦⇒⟧ R
⟦lam⟧ {P} η p = λ wk q → η (ren-P wk p , q)
  where open Sem P renaming (ren to ren-P)

⟦app⟧ : ∀ {P Q R} → P → Q ⟦⇒⟧ R → P → Q → P → R
⟦app⟧ η μ = λ px → η px id (μ px)
```

Remark: The above construction of the exponential is taken from MacLane & Moerdijk's Sheaves in Geometry and Logic (p.46).

At this stage, we have enough structure to interpret the types:

```
⟦_⟧ : type → Sem
⟦ unit ⟧  = ⟦unit⟧
⟦ S ⇒ T ⟧ = ⟦ S ⟧ ⟦⇒⟧ ⟦ T ⟧
⟦ A * B ⟧ = ⟦ A ⟧ ⟦×⟧ ⟦ B ⟧
```

To interpret contexts, we need a terminal object:

```
T^: Sem
T^= record { _⊢ = λ _ → ⊤
           ; ren = λ _ _ → tt }

⟦_⟧C : (Γ : context) → Sem
⟦ ε ⟧C     = T^
⟦ Γ ▹ T ⟧C = ⟦ Γ ⟧C ⟦×⟧ ⟦ T ⟧
```

As usual, a type in context will be interpreted as a morphism between their respective interpretations. The interpreter then takes the syntactic object to its semantical counterpart:

```
_□_ : context → type → Set
Γ □ T = ⟦ Γ ⟧C → ⟦ T ⟧

lookup : ∀ {Γ T} → T ∈ Γ → Γ □ T
lookup here (_ , v)      = v
lookup (there x) (γ , _) = lookup x γ

eval : ∀{Γ T} → Γ ⊢ T → Γ □ T
eval {Γ} (lam {S}{T} b)    = ⟦lam⟧ {⟦ Γ ⟧C}{⟦ S ⟧}{⟦ T ⟧} (eval b)
eval (var v)               = lookup v
eval {Γ}{T} (_!_ {S} f s)  = ⟦app⟧ {⟦ Γ ⟧C}{⟦ S ⟧}{⟦ T ⟧} (eval f) (eval s)
eval {Γ} tt                = ⟦tt⟧ {⟦ Γ ⟧C}
eval {Γ} (pair {A}{B} a b) = ⟦pair⟧ {⟦ Γ ⟧C}{⟦ A ⟧}{⟦ B ⟧} (eval a) (eval b)
eval {Γ} (fst {A}{B} p)    = ⟦fst⟧ {⟦ Γ ⟧C}{⟦ A ⟧}{⟦ B ⟧} (eval p)
eval {Γ} (snd {A}{B} p)    = ⟦snd⟧ {⟦ Γ ⟧C}{⟦ A ⟧}{⟦ B ⟧} (eval p)
```

Reify and reflect are defined at a given syntactic context, we therefore introduce suitable notations:

```
[_]□_ : context → type → Set
[ Γ ]□ T = Γ ⊢⟦T⟧
  where open Sem ⟦ T ⟧ renaming (_⊢ to _⊢⟦T⟧)

[_]□C_ : context → context → Set
[ Γ ]□C Δ = Γ ⊢⟦Δ⟧C
  where open Sem ⟦ Δ ⟧C renaming (_⊢ to _⊢⟦Δ⟧C)
```

The sea has sufficiently risen: we can implement our initial plan, using the renaming operator `ren` equipping `Sem` in the function case in `reify`:

```
reify : ∀ {T Γ} → [ Γ ]□ T  → Γ ⊢Nf T
reflect : ∀ {Γ} → (T : type) → Γ ⊢Ne T → [ Γ ]□ T

reify {unit} v        = v
reify {A * B} (a , b) = pair (reify a) (reify b)
reify {S ⇒ T} f       = lam (reify (app {S}{T} (ren (weak1 id) f) (reflect S (var here))))
  where open Sem ⟦ S ⇒ T ⟧

        app : ∀{S T Γ} → [ Γ ]□ (S ⇒ T) → [ Γ ]□ S → [ Γ ]□ T
        app f s = f id s

reflect unit v    = ground v
reflect (A * B) v = reflect A (fst v) , reflect B (snd v)
reflect (S ⇒ T) v = λ w s → reflect T (ren w v ! reify s)
  where open Sem (Nê (S ⇒ T))
```

We generalize `reify` to work on any "term in an environement", using the identity context, from which we obtain the normalization function:

```
reify-id : ∀{Γ T} → Γ □ T → Γ ⊢Nf T
reify-id {Γ}{T} f = reify (f (idC Γ))
  where open Sem

        idC : ∀ Γ → [ Γ ]□C Γ
        idC ε       = tt
        idC (Γ ▹ T) = ren ⟦ Γ ⟧C (weak1 id) (idC Γ) , reflect T (var here)
```

```
norm : ∀{Γ T} → Γ ⊢ T → Γ ⊢Nf T
norm = reify-id ∘ eval
```

Remark: For pedagogical reasons, we have defined `reify {S ⇒ T} f` using function application and weakening, without explicitly using the structure of `f : [ Γ ]□ S ⟦⇒⟧ T`. However, there is also a direct implementation:

```
remark-reify-fun : ∀ {Γ S T} → (f : [ Γ ]□ (S ⇒ T)) →
    reify {S ⇒ T} f ≡ lam (reify (f (weak1 id) (reflect S (var here))))
remark-reify-fun f = refl
```

CHAPTER

FOUR

CONCLUSION

In the first and second part, we have seen how inductive types and families can be used to build initial models, supporting the definition of various interpretations in Agda itself.

In the third part, we have seen how, by defining a richly-structured model, we could implement a typed model of the λ-calculus and manipulate binders in the model.

Exercises (difficulty: open ended):

1. Integrate the results from last week with this week's model of the λ-calculus so as to quotient this extended calculus. Hint: have a look at Normalization by evaluation and algebraic effects
2. The models we have constructed combine a semantical aspect (in Agda) and a syntactic aspect (judgments `_⊢Nf_`). This has been extensively studied under the name of "glueing". We took this construction as granted here.
3. Prove the correctness of the normalisation function `norm`. The categorical semantics (described in the next section) provides the blueprint of the necessary proofs.

CHAPTER

FIVE

OPTIONAL: CATEGORICAL SPOTTING

We have been using various categorical concepts in this lecture. For the sake of completeness, we (partially) formalize these notions in Agda with extensional equality.

Remark: The point of this exercise is **certainly not** to define category theory in type theory: this would be, in my opinion, a pointless exercise (from a pedagogical standpoint, anyway). Rather, we merely use the syntactic nature of type theory and our computational intuition for it to provide a glimpse of some categorical objects (which are much richer than what we could imperfectly model here!).

First, we model the notion of category:

```
record Cat : Set where
 field
    Obj : Set
    Hom[_:_] : Obj → Obj → Set
    idC : ∀ {X} → Hom[ X : X ]
    _∘C_ : ∀ {X Y Z} → Hom[ Y : Z ] → Hom[ X : Y ] → Hom[ X : Z ]

record IsCat (C : Cat) : Set where
  open Cat C
  field
    left-id : ∀ {A B} → ∀ (f : Hom[ A : B ]) →
              idC ∘C f ≡ f
    right-id : ∀ {A B} → ∀ (f : Hom[ A : B ]) →
               f ∘C idC ≡ f
    assoc : ∀ {A B C D} → ∀ (f : Hom[ A : B ])(g : Hom[ B : C ])(h : Hom[ C : D ]) →
            (h ∘C g) ∘C f ≡ h ∘C (g ∘C f)
```

Contexts form a category, hence the emphasis we have put on defining composition of weakenings:

```
Context-Cat : Cat
Context-Cat = record { Obj = context ;
                       Hom[_:_] = _≥_ ;
                       idC = id ;
                       _∘C_ = _∘wk_ }
```

Our model of semantics objects is actually an instance of a more general object, called a "presheaf", and defined over any category as the class of functors from the opposite category of `C` to `Set`:

```
record PSh (C : Cat) : Set₁ where
  open Cat C
  field
    _⊢ : Obj → Set
    ren : ∀ {X Y} → Hom[ X : Y ] → Y ⊢ → X ⊢

record IsPSh {C : Cat}(P : PSh C) : Set where
```

```
  open Cat C
  open PSh P
  field
    ren-id : ∀ {X} → ren (idC {X}) ≡ λ x → x
    ren-∘ : ∀ {X Y Z x} → (g : Hom[ Y : Z ])(f : Hom[ X : Y ]) →
            ren (g ∘C f) x ≡ ren f (ren g x)
```

A presheaf itself is a category, whose morphisms are natural transformations:

```
Hom[_][_:_] : ∀ (C : Cat)(P : PSh C)(Q : PSh C) → Set
Hom[ C ][ P : Q ] = ∀ {Γ} → Γ ⊢P → Γ ⊢Q
  where open PSh P renaming (_⊢ to _⊢P)
        open PSh Q renaming (_⊢ to _⊢Q)
        open Cat C

record IsPShHom {C P Q}(η : Hom[ C ][ P : Q ]) : Set where
  open Cat C
  open PSh P renaming (_⊢ to _⊢P ; ren to ren-P)
  open PSh Q renaming (_⊢ to _⊢Q ; ren to ren-Q)
  field
    naturality : ∀ {Γ Δ}(f : Hom[ Γ : Δ ])(x : Δ ⊢P) →
                 η (ren-P f x) ≡ ren-Q f (η x)

PSh-Cat : Cat → Cat
PSh-Cat C = record { Obj = PSh C
                   ; Hom[_:_] = λ P Q → Hom[ C ][ P : Q ]
                   ; idC = λ x → x
                   ; _∘C_ = λ f g x → f (g x) }

PSh-IsCat : (C : Cat) → IsCat (PSh-Cat C)
PSh-IsCat C = record { left-id = λ f → refl
                     ; right-id = λ f → refl
                     ; assoc = λ f g h → refl }
```

Remark: We have been slightly cavalier in the definition of `PSh-Cat`: we ought to make sure that the objects in `Obj` do indeed satisfy `IsPSh` whereas the morphisms in `Hom[_:_]` do indeed satisfy `IsPShHom`. However, these are not necessary to prove that presheaves form a category so we eluded them here, for simplicity.

Our definition of `Sem` thus amounts to `PSh[context]`:

```
PSh[context] = PSh Context-Cat
```

The `Y` operator is a general construction, called the Yoneda lemma. Given any *function* `F : context → Set`, the Yoneda embedding gives us the ability to produce a presheaf from **any** function:

```
Yoneda : (F : context → Set) → PSh[context]
Yoneda F = record { _⊢ = λ Γ → ∀ {Δ} → Hom[ Δ : Γ ] → F Δ
                  ; ren = λ wk₁ k wk₂ → k (wk₁ ∘wk wk₂) }
       where open Cat Context-Cat

Yoneda-IsPSh : {F : context → Set} → IsPSh (Yoneda F)
Yoneda-IsPSh = record { ren-id = λ {X} → ext λ ρ →
                                         ext-impl (λ Γ →
                                         ext λ wk →
                                         cong (ρ {Γ}) (lemma-left-unit wk))
                      ; ren-∘ = λ {Δ}{∇}{Ω}{k} wk₁ wk₂ →
                                         ext-impl λ Γ →
                                         ext λ wk₃ →
```

```
cong k (lemma-assoc wk₃ wk₂ wk₁) }
```

A precise treatment of the categorical aspects of normalization-by-evaluation for the λ-calculus can be found in the excellent Normalization and the Yoneda embedding or, in a different style, in Semantics Analysis of Normalisation by Evaluation for Typed Lambda Calculus.