Total Programming

Release 1.0

Pierre-Évariste Dagand

Feb 08, 2018

CONTENTS

1 First-order Unification 3
1.1 Specification: terms 3
1.2 Specification: occur-check 3
1.3 Specification: substitution 4
1.4 Specification: most-general unifier 5
1.5 Structurally: terms 6
1.6 Structurally: variable extrusion 6
1.7 Structurally: occur-check 7
1.8 Structurally: single term substitution 8
1.9 Structurally: substitution 8
1.10 Structurally: most-general unifier 8

2 Proof search 11
2.1 Specification 11
2.2 Structural search 12
2.3 Compact search 13
2.4 Interlude: induction / memoisation 14
2.5 Terminating search 16

3 General recursion 19
3.1 Syntax for general recursion 19
3.2 Monad morphism 20
3.3 Interpretation: identity 20
3.4 Interpretation: immediate values 20
3.5 Interpretation: step-indexing 21
3.6 Interlude: Universe of (collapsible) predicates 21
3.7 Collapsible accessibility predicate 22
3.8 Postlude: collapsible, formally 23

Last week:

- use indices to garantee totality
- definitions by induction over (faithful) syntax
- one shade of salmon: structural substitution

Today, we shall:

- study non-obviously terminating programs
- argue about termination *against* Agda's termination checker
- argue about termination *with* Agda's typechecker

Outline:

1. First-order unification: using indices to provide order
2. Proof search: using induction to witness termination
3. Bove-Capretta: using monads to postpone The Talk about termination

The vision: Total Functional Programming

- zealously pursued with dependent types by Conor McBride
- at the origin of *all* of today's examples

CHAPTER

ONE

FIRST-ORDER UNIFICATION

In this first example, we set out to implement first-order unification. We first implement/specify the problem assuming general recursion and then set out to re-implement it in a structurally recursive manner. This presentation was given by McBride in First-order unification by structural recursion.

1.1 Specification: terms

We study a simple syntax of terms with variables `var` (represented by natural numbers), a nullary constructor `leaf` and a binary constructor `fork`:

```
Var = ℕ

data Term : Set where
  var : (i : Var) → Term
  leaf : Term
  fork : (s t : Term) → Term
```

One can easily generalize this to any term signature but this would needlessly pollute our definitions. Crucially, all the definitions below will behave *functorially* over `leaf` and `fork`.

Indeed, `Term` is a free term algebra! It therefore comes with a simultaneous substitution:

```
sub : (Var → Term) → Term → Term
sub ρ (var i) = ρ i
sub ρ leaf = leaf
sub ρ (fork s t) = fork (sub ρ s) (sub ρ t)

_∘K_ : (Var → Term) → (Var → Term) → Var → Term
ρ₁ ∘K ρ₂ = λ k → sub ρ₁ (ρ₂ k)
```

In the first lecture, the function `sub` was called `bind` but it is otherwise exactly the same thing.

1.2 Specification: occur-check

For a variable `i : Var` and a term `t : Term`, there are two cases for the occur-check `check i t`:

- either `i` occurs in `t`,
- or `i` does not occur in `t`

In the latter case, the term can be made “smaller” by noticing that, since `i` does not occur in `t`, we can rename every variables `j > i` to `j - 1` while leaving all `j < i` unmodified.

To this end, we implement a test-and-reduce operator `i Δ j` on variables `i, j : Var` which raises if the variables are equal (which will be interpreted as a successful occur-check) and returns the renamed `j` otherwise:

```
_Δ_ : Var → Var → Maybe Var
zero  Δ zero  = ∅
zero  Δ suc y = return y
suc x Δ zero  = return zero
suc x Δ suc y = x Δ y >>= λ y' →
                return (suc y')
```

The occur-check consists then simply in applying Δ to every variable of the given term, unless `i` actually occurs in which case it will raise:

```
check : Var →  Term → Maybe Term
check i (var j)    = i Δ j >>= λ z →
                     return (var z)
check i leaf       = return leaf
check i (fork f s) = check i f >>= λ r1 →
                     check i s >>= λ r2 →
                     return (fork r1 r2)
```

When `check i t` succeeds in producing a term `t'`, we thus now that `i` did not occur in `t` and that all variables above `i` have been lowered by 1 in `t'`.

1.3 Specification: substitution

We have seen that terms are endowed with a substitution operator that expects a function of type `Var → Term`. The role of first-order unification is to compute such a substitution. However, the kind of objects we will be building contains much more structure that the rather bland function space `Var → Term`.

So, as usual, we make a detour through a precise and inductive characterization of the mapping from terms to variables that we shall consider:

```
data Subst : Set where
  id : Subst
  _∷[_/_] : (σ : Subst)(t : Term)(i : Var) → Subst
```

In turn, we interpret this initial model in the target one. Substituting a single term `t : Term` for a variable `i : Var` amounts to a substitution that returns `t` if `i ≟ j`, and the remainder of `j` by `i` otherwise:

```
_for_ : Term → Var → (Var → Term)
(t for i) j with i Δ j
... | nothing = t
... | just j' = var j'
```

The interpretation of a list of single substitutions is merely function composition:

```
⟦_⟧ : Subst → (Var → Term)
⟦ id ⟧ = var
⟦ ρ ∷[ t / i ] ⟧ = ⟦ ρ ⟧ ∘K (t for i)
```

1.4 Specification: most-general unifier

The computation of the most-general unifier works by accumulating a substitution as it explores matching subterms (case `amgu (fork s₁ t₁) (fork s₂ t₂)`) and then discharging that substitution (case `amgu s t (σ ::[ r / z ])`). Variables are only considered under no substitution (cases `amgu _ _ id`), in which case we must either solve a flex-flex problem or a flex-rigid problem:

```
flex-flex : (x y : Var) → Subst
flex-rigid : (x : Var)(t : Term) → Maybe Subst

{-# TERMINATING #-}
amgu : (s t : Term)(acc : Subst) → Maybe Subst
-- Conflicts:
amgu leaf (fork _ _) _             = ∅
amgu (fork _ _) leaf _             = ∅
-- Matches:
amgu leaf leaf acc                 = return acc
amgu (fork s₁ t₁) (fork s₂ t₂) acc = amgu s₁ s₂ acc >>= λ acc →
                                     amgu t₁ t₂ acc
-- Variables flex-flex:
amgu (var x) (var y) id            = return (flex-flex x y)
-- Variables flex-rigid:
amgu (var x) t id                  = flex-rigid x t
amgu t (var x) id                  = flex-rigid x t
-- Terms under substitution:
amgu s t (σ ::[ r / z ])           = amgu (sub (r for z) s)
                                          (sub (r for z) t) σ >>= λ σ →
                                     return (σ ::[ r / z ])

flex-flex x y with x Δ y
... | just y' = id ::[ var y' / x ]
... | nothing = id

flex-rigid x t = check x t >>= λ t' →
                 return (id ::[ t' / x ])

mgu : (s t : Term) → Maybe Subst
mgu s t = amgu s t id
```

Assuming that the above definition is terminating, we can test it on a few examples:

```
test₁ : mgu (fork v₀ leaf) (fork (fork leaf leaf) v₁)
      ≡ just (id ::[ leaf / 0 ] ::[ (fork leaf leaf) / 0 ])
test₁ = refl

test₂ : mgu (fork v₀ leaf) (fork (fork leaf leaf) v₃)
      ≡ just (id ::[ leaf / 2 ] ::[ (fork leaf leaf) / 0 ])
test₂ = refl

test₃ : mgu v₀ (fork leaf v₀)
      ≡ nothing
test₃ = refl

test₄ : mgu (fork v₀ leaf) (fork (fork leaf leaf) v₀)
      ≡ nothing
test₄ = refl
```

```
test₅ : mgu (fork v₀ v₁) (fork (fork leaf v₁) (fork leaf leaf))
        ≡ just (id ::[ fork leaf leaf / 0 ] ::[ fork leaf (var zero) / 0 ])
test₅ = refl
```

1.5 Structurally: terms

As it stands, we will have a hard time convincing Agda that this implementation is indeed terminating: the terms grow as substitutions are discharged while the accumulated substitution itself grows as flex-rigid are solved.

Part of the problem stands in the fact that, whilst we have the intuition that the numbers of variables occuring in terms keeps decreasing as unification proceeds, this intuition is not documented in the code. Let us try again, using indexing as a machine-checked mode of documentation.

We now stratify the set of variables, ie. `Var n` contains `n` distinct variables:

```
Var : ℕ → Set
Var = Fin
```

We can thus represent *terms with (at most) ‘‘n‘‘ variables*:

```
data Term (n : ℕ) : Set where
  var : (i : Var n) → Term n
  leaf : Term n
  fork : (s t : Term n) → Term n
```

Exercise (difficulty: 1) Once again, we can implement substitution:

```
sub : ∀ {m n} → (Var m → Term n) → Term m → Term n
sub ρ t = {!!}

_∘K_ : ∀ {m n l} → (Var m → Term n) → (Var l → Term m) → Var l → Term n
ρ₁ ∘K ρ₂ = {!!}
```

Exercise (difficulty: 1) Implement the (obvious) renaming operation:

```
ren : ∀ {m n} → (Var m → Var n) → Term m → Term n
ren σ t = {!!}
```

Remark: Two substitutions are equal if they are equal pointwise:

```
_≐_ : ∀ {m n} → (f g : Var m → Term n) → Set
f ≐ g = ∀ x → f x ≡ g x
```

1.6 Structurally: variable extrusion

Variable comparison becomes more informative for Agda since we can witness in the return type that the variable `y` was definitely distinct from `x` and, therefore, belongs to a strictly smaller class of variables:

```
_△_ : ∀ {n} → Var (suc n) → Var (suc n) → Maybe (Var n)
zero △ zero                  = ∅
zero △ suc y                 = return y
_△_ {zero} (suc ())
```

```
_Δ_ {suc _} (suc x) zero    = return zero
_Δ_ {suc _} (suc x) (suc y) = x Δ y >>= λ y' →
                             return (suc y')
```

Exercise (difficulty: 1) The operation Δ can be understood as the partial inverse of the following injection from `Var n` to `Var (suc n)` which adds `i` to the variables in `Var n`:

```
inj[_] : ∀ {n} → (i : Var (suc n)) → Var n → Var (suc n)
inj[ zero ] y = suc y
inj[ suc x ] zero = zero
inj[ suc x ] (suc y) = suc (inj[ x ] y)
```

Prove the following lemmas, the last being one way to state that `inj[_]` is the partial inverse of Δ:

```
lemma-inj1 : ∀ {n} x y z → inj[_] {n} x y ≡ inj[_] x z → y ≡ z
lemma-inj1 = {!!}

lemma-inj2 : ∀ {n} x y → inj[_] {n} x y ≢ x
lemma-inj2 = {!!}

lemma-inj3 : ∀ {n} x y → x ≢ y → ∃ λ y' → inj[_] {n} x y' ≡ y
lemma-inj3 = {!!}

lemma-inj-Δ : ∀ {n}(x y : Var (suc n))(r : Maybe (Var n)) →
  x Δ y ≡ r → ((y ≡ x × r ≡ nothing) ⊎ (∃ λ y' → y ≡ inj[ x ] y' × r ≡ just y'))
lemma-inj-Δ = {!!}
```

Another way to construct Δ is to obtain it as a view (`inj-view` is essentially a proof-carrying version of Δ):

```
data inj-View {n}(i : Var (suc n)) : Var (suc n) → Set where
  just : (k : Var n) → inj-View i (inj[ i ] k)
  eq : inj-View i i

inj-view : ∀ {n}(i : Var (suc n))(j : Var (suc n)) → inj-View i j
inj-view i j = {!!}
```

1.7 Structurally: occur-check

Following Δ, the occur-check reflects the fact that, in case of success, the resulting term did not use one variable:

```
check : ∀ {n} → (i : Var (suc n))(t : Term (suc n)) → Maybe (Term n)
check i (var j)    = i Δ j >>= λ k →
                     return (var k)
check i leaf       = return leaf
check i (fork f s) = check i f >>= λ r1 →
                     check i s >>= λ r2 →
                     return (fork r1 r2)
```

If we were able to extrude `x` from `t` into `t'`, this means that injecting `x` into `t'` amounts to the exact same term `t`:

```
lemma-check : ∀ {n} x t {t'} → check {n} x t ≡ just t' → ren (inj[ x ]) t' ≡ t
lemma-check x y p = {!!}
```

1.8 Structurally: single term substitution

Crucially, a (single) substitution ensures that a variable denotes a term with one less variable:

```
_for_ : ∀ {n} → Term n → Var (suc n) → (Var (suc n) → Term n)
(t' for x) y with x Δ y
... | just y' = var y'
... | nothing = t'
```

The composition of `_for_` and `inj[_]` amounts to an identity:

```
lemma-for-inj : ∀ {n} (t : Term n) x → ((t for x) ∘ (inj[ x ])) ≐ var
lemma-for-inj = {!!}

lemma-check-inj : ∀ {n} x t t' → check {n} x t ≡ just t' →
  sub (t' for x) t  ≡ sub (t' for x) (var x)
lemma-check-inj = {!!}
```

1.9 Structurally: substitution

Iteratively, a substitution counts the upper-bound of variables:

```
data Subst : ℕ → ℕ → Set where
  id : ∀ {n} → Subst n n
  _∷[_/_] : ∀ {m n} → (σ : Subst m n)(t' : Term m)(x : Var (suc m)) → Subst (suc m) n

⟦_⟧ : ∀ {m n} → Subst m n → (Var m → Term n)
⟦_⟧ id = var
⟦_⟧ (ρ ∷[ t' / x ]) = ⟦ ρ ⟧ ∘K (t' for x)
```

Exercise (difficulty: 1) Implement composition on the inductive characterization of substitutions and show that it corresponds to the underlying composition of substitutions:

```
_∘A_ : ∀ {l m n} → Subst m n → Subst l m → Subst l n
ρ ∘A σ = {!!}

lemma-comp : ∀ {l m n} (ρ : Subst m n)(σ : Subst l m) → ⟦ ρ ∘A σ ⟧ ≡ ⟦ ρ ⟧ ∘K ⟦ σ ⟧
lemma-comp = {!!}
```

1.10 Structurally: most-general unifier

The implementation of the most-general unifier is exactly the same, excepted that termination has become self-evident: when performing the substitution (case `amgu {suc k} _ _ (m , (σ ∷[ r / z ]))`), the next call to `amgu` will be on terms with `k < suc k` variables. It is therefore definable by structural recursion and Agda is able to spot it:

```
flex-flex : ∀ {m} → (x y : Var m) → ∃ (Subst m)
flex-rigid : ∀ {m} → (x : Var m)(t : Term m) → Maybe (∃ (Subst m))

amgu : ∀ {m} → (s t : Term m)(acc : ∃ (Subst m)) → Maybe (∃ (Subst m))
-- Conflicts:
amgu leaf (fork _ _) _ = ∅
amgu (fork _ _) leaf _ = ∅
-- Matches:
amgu leaf leaf acc = return acc
amgu (fork s₁ t₁) (fork s₂ t₂) acc = amgu s₁ s₂ acc >>= λ acc →
                                     amgu t₁ t₂ acc
-- Variables flex-flex:
amgu (var x) (var y) (m , id) = return (flex-flex x y)
-- Variables flex-rigid:
amgu (var x) t (m , id) = flex-rigid x t
amgu t (var x) (m , id) = flex-rigid x t
-- Terms under substitution:
amgu {suc k} s t (m , (σ ∷[ r / z ])) =
  amgu {k} (sub (r for z) s)
           (sub (r for z) t) (m , σ) >>= λ { (n , σ) →
  return ((n , σ ∷[ r / z ])) }

flex-flex {zero} ()
flex-flex {suc _} x y with x Δ y
... | just y' = , id ∷[ var y' / x ]
... | nothing = , id

flex-rigid {0} ()
flex-rigid {suc _} x t = check x t >>= λ t' →
                         return (, id ∷[ t' / x ])


mgu : ∀ {m} → (s t : Term m) → Maybe (∃ (Subst m))
mgu s t = amgu s t (, id)
```

The key idea was thus to reify the (decreasing) *measure* as an indexing discipline. Our implementation was then naturally defined structurally over this index, thus yielding a structurally acceptable definition.

Exercise (difficulty: 3) Prove the *soundness* of your implementation: the substitution thus computed is indeed a valid unifier. The lemmas left as exercises will be useful there.

Exercise (difficulty: 5) Prove the *completeness* if your implementation: the substitution thus computed is indeed the most general one. You may want to invest into some archaeological investigation or have a look at the literature such as, for example, Type inference in context.

CHAPTER
TWO

PROOF SEARCH

In this second example, we study a decision procedure studied by Roy Dyckhoff in Contraction-free sequent calculi for intuitionistic logic and turned into type theory by Conor McBride in Djinn, monotonic.

2.1 Specification

We consider the purely negative fragment of propositional logic:

```
data Formula : Set where
  Atom : (a : A) → Formula
  _⊃_ : (P Q : Formula) → Formula
```

The decision procedure checks whether a Formula (in a context) is true. This amounts to implementing a traditional focusing presentation of the sequent calculus:

```
{-# TERMINATING #-}
_⊢_ : List Formula → Formula → Bool
_[_]⊢_ : List Formula → Formula → A → Bool
_><_⊢ax_ : Bwd Formula → Fwd Formula → A → Bool

Γ ⊢ P ⊃ Q         = (Γ ▹ P) ⊢ Q
Γ ⊢ Atom a        = ϵ >< Γ ⊢ax a

Δ >< ϵ       ⊢ax α = false
Δ >< (P ◃ Γ) ⊢ax α = (Δ ++ Γ) [ P ]⊢ α
                  ∨ (Δ ▹ P) >< Γ ⊢ax α

Γ [ Atom α ]⊢ β    = ⌊ α ≟ β ⌋
Γ [ P ⊃ Q ]⊢ α     = Γ [ Q ]⊢ α ∧ Γ ⊢ P

⊢_ : Formula → Bool
⊢ P = [] ⊢ P
```

This definition is terminating but not obviously so. The crux of the matter is in `_><_⊢ax_`, which reduces the context on one hand (call `(Δ ++ Γ) [ P ]⊢ α`) while `_⊢_` called from `_[_]⊢_` will augment the context.

Here are a few tests:

```
test₁ : ⊢ A ⊃ B ⊃ A ≡ true
test₁ = refl

test₂ : ⊢ A ⊃ B ≡ false
test₂ = refl
```

```
CPS : Formula → Formula
CPS A = (A ⊃ ∐) ⊃ ∐

return : ⊢ A ⊃ CPS A ≡ true
return = refl

bind : ⊢ CPS A ⊃ (A ⊃ CPS B) ⊃ CPS B ≡ true
bind = refl

call-cc : ⊢ ((A ⊃ CPS B) ⊃ CPS A) ⊃ CPS A ≡ true
call-cc = refl
```

2.2 Structural search

Following the lesson from the first part, we turn the ordering, which justifies our definition, into an indexing discipline. Despite the fact that the context shrinks then grows, an important observation is that, when a formula is taken out of the context, the formuli that may be subsequently inserted are necessarily its premises, of *strictly lower order*. We thus capture the (upper-bound) order of formuli by a suitable indexing strategy:

```
data Formula : ℕ → Set where
  Atom : ∀ {n} → (a : A) → Formula n
  _⊃_ : ∀ {n} → (P : Formula n)(Q : Formula (suc n)) → Formula (suc n)
```

The representation of context also needs to be stratified, so that formulis come up sorted along their respective order:

```
Bucket : Set → Set
Bucket X = Σ[ n ∈ ℕ ] (Vec X n)

Context : ℕ → Set
Context 0 = ⊤
Context (suc n) = Bucket (Formula n) × Context n
```

Exercise (difficulty: 1) Implement the usual operations of a context/list:

```
[]C : ∀ {n} → Context n
[]C = {!!}

infixl 70 _▹C_
_▹C_ : ∀ {n} → Context (suc n) → Formula n → Context (suc n)
_▹C_ = {!!}

_++C_ : ∀ {n} → Context n → Context n → Context n
_++C_ = {!!}
```

With a bit of refactoring, we can integrate indices as well as absorb the zipper traversal, making the structural recursion slightly more obvious (to us, not to Agda):

```
pick1 : ∀ {X : Set}{n} → Vec X n → Vec (X × Vec X (pred n)) n
pick1 {X} xs = help [] xs []
  where help : ∀ {k l} → Vec X k → Vec X l
                       → Vec (X × Vec X (pred (k + l))) k
                       → Vec (X × Vec X (pred (k + l))) (k + l)
```

```
        help Δ []  acc = acc
        help Δ (P ∷ Γ) acc = help (Δ ▹ P) Γ ((P , Δ ++ Γ) ∷ acc)

any : ∀ {n} ⟶ Vec Bool n ⟶ Bool
any [] = false
any (false ∷ xs) = any xs
any (true ∷ xs) = true


{-# TERMINATING #-}
_/_⊢_ : ∀ {n l} ⟶ Vec (Formula (suc n)) l ⟶ Context (suc n) ⟶ Formula n ⟶ Bool
_/_[_]⊢_ : ∀ {n l} ⟶ Vec (Formula n) l ⟶ Context n ⟶ Formula n ⟶ A ⟶ Bool
search : ∀ {n} ⟶ Context n ⟶ A ⟶ Bool

B / Γ      ⊢ Atom α      = search ((, B) , Γ) α
B / B₂ , Γ ⊢ P ⊃ Q      = B / B₂ , Γ ▹C P  ⊢ Q

B / Γ [ Atom α ]⊢ β      = ⌊ α ≟ β ⌋
B / Γ [ P ⊃ Q  ]⊢ β      = B / Γ [ Q ]⊢ β ∧ B / Γ ⊢ P

search {zero} tt α = false
search {suc n} ((l , B) , Γ) α =
  let try = map (λ { (P , B) ⟶ B / Γ [ P ]⊢ α })
                (pick1 B)
  in
  any try ∨ search Γ α

⊢_ : Formula 42 ⟶ Bool
⊢_ P = [] / []C ⊢ P
```

2.3 Compact search

The previous implementation was needlessly mutually recursive. We inline (at the expense of clarity, sadly) the purely structural definitions on `Formulas`:

```
{-# TERMINATING #-}
search : ∀ {n} ⟶ Context n ⟶ A ⟶ Bool
search {zero} tt α = false
search {suc n} ((l , B) , Γ) α =
  let try = map (λ { (P , B) ⟶ B / Γ [ P ]⊢ α })
                (pick1 B)
  in
  any try ∨ search Γ α
    where _/_[_]⊢_ : Vec (Formula n) (pred l) ⟶ Context n ⟶ Formula n ⟶ A ⟶ Bool
          B / Γ [ Atom α ]⊢ β = ⌊ α ≟ β ⌋
          B / Γ [ _⊃_ {n} P Q  ]⊢ β = B / Γ [ Q ]⊢ β ∧ B / Γ ⊢ P
            where  _/_⊢_ : Vec (Formula (suc n)) (pred l) ⟶ Context (suc n) ⟶ Formula n ⟶ Bool
                   B / Γ ⊢ Atom α = search ((, B) , Γ) α
                   B / B₂ , Γ ⊢ P ⊃ Q  = B / B₂ , Γ ▹C P  ⊢ Q

_⊢_ : ∀ {n} ⟶ Context n ⟶ Formula n ⟶ Bool
Γ ⊢ Atom α = search Γ α
Γ ⊢ P ⊃ Q  = Γ ▹C P  ⊢ Q

⊢_ : Formula 42 ⟶ Bool
```

```
⊢_ P = []C ⊢ P
```

Once again, termination becomes clearer for us but still out of Agda's grasp.

2.4 Interlude: induction / memoisation

The Coq layman tends to see induction principles as a reassuring meta-theoretical objects which is automatically produced by Coq when `Inductive` is invoked but never actually used by the user, who resorts to `match (..) with (..)` in programs or the `induction` tactics in proofs. The Agda layman just knows that dependent pattern-matching could in principle be expressed with induction principles (Pattern Matching in Type Theory, Eliminating Dependent Pattern Matching) and, therefore, that all is meta-theoretically fine.

With The View from the Left came the idea that one could get the benefits of pattern-matching *syntax* while actually appealing to induction principles to back them up *semantically*.

Assuming that we had this machinery (which we have not), it becomes interesting to study and develop the algebra of induction principles. Let us dissect the induction principle for natural numbers.

The first ingredient of an induction principle is the *induction hypothesis*. We can generically define an induction hypothesis as a predicate transformer computing the necessary hypothesis:

```
RecStruct : Set → Set₁
RecStruct A = (A → Set) → (A → Set)

Rec-ℕ : RecStruct ℕ
Rec-ℕ P zero    = ⊤
Rec-ℕ P (suc n) = P n
```

Assuming that we have established the *induction step*, we ought to be able to prove any induction hypothesis:

```
RecursorBuilder : ∀ {A : Set} → RecStruct A → Set₁
RecursorBuilder Rec = ∀ P → (∀ a → Rec P a → P a) → ∀ a → Rec P a

rec-ℕ-builder : RecursorBuilder Rec-ℕ
rec-ℕ-builder P f zero    = tt
rec-ℕ-builder P f (suc n) = f n (rec-ℕ-builder P f n)
```

Therefore, typing the knot, given an induction step, we ought to be able to establish the desired predicate:

```
Recursor : ∀ {A : Set} → RecStruct A → Set₁
Recursor Rec = ∀ P → (∀ a → Rec P a → P a) → ∀ a → P a

build : ∀ {A : Set} {Rec : RecStruct A} →
        RecursorBuilder Rec → Recursor Rec
build builder P f x = f x (builder P f x)

rec-ℕ : Recursor Rec-ℕ
rec-ℕ = build rec-ℕ-builder
```

These recursors have trivial "terminal" object, which amount to performing no induction at all (as well we shall see, it has its uses, like the unit type):

```
Rec-1 : ∀ {X : Set} → RecStruct X
Rec-1 P x = ⊤
```

```
rec-1-builder : ∀ {X} → RecursorBuilder (Rec-1 {X})
rec-1-builder P f x = tt
```

More interestingly, we can define induction on pairs by (arbitrarily) deciding that the first element must be strictly decreasing. In effect, this is what we do when manipulating `Bucket`, asking only for the size of the underlying vector to decrease:

```
Σ1-Rec : ∀ {A : Set}{B : A → Set} →
        RecStruct A →
        RecStruct (Σ A B)
Σ1-Rec RecA P (x , y) =
  RecA (λ x' → ∀ y' → P (x' , y')) x

Rec-Bucket : ∀ {X} → RecStruct (Bucket X)
Rec-Bucket  = Σ1-Rec Rec-ℕ

Σ1-rec-builder : ∀ {A : Set}{B : A → Set}{RecA : RecStruct A} →
  RecursorBuilder RecA → RecursorBuilder (Σ1-Rec {A = A}{B = B} RecA)
Σ1-rec-builder {RecA = RecA} recA P f (x , y) =
  recA _ (λ a a-rec b → f (a , b) a-rec) x

rec-Bucket-builder : ∀ {X} → RecursorBuilder (Rec-Bucket {X})
rec-Bucket-builder {X} = Σ1-rec-builder rec-ℕ-builder
```

In fact, this latter recursor is a special case of a powerful recursion structure, lexicographic recursion:

```
Σ-Rec : ∀ {A : Set}{B : A → Set} →
        RecStruct A → (∀ x → RecStruct (B x)) →
        RecStruct (Σ A B)
Σ-Rec RecA RecB P (x , y) =
  -- Either x is constant and y is "smaller", ...
  RecB x (λ y' → P (x , y')) y
  ×
  -- ...or x is "smaller" and y is arbitrary.
  RecA (λ x' → ∀ y' → P (x' , y')) x

Σ-rec-builder :
  ∀ {A : Set} {B : A → Set}
  {RecA : RecStruct A}
  {RecB : ∀ x → RecStruct (B x)} →
  RecursorBuilder RecA → (∀ x → RecursorBuilder (RecB x)) →
  RecursorBuilder (Σ-Rec RecA RecB)
Σ-rec-builder {RecA = RecA} {RecB = RecB} recA recB P f (x , y) =
  (p₁ x y p₂x , p₂x)
    where
      p₁ : ∀ x y →
           RecA (λ x' → ∀ y' → P (x' , y')) x →
           RecB x (λ y' → P (x , y')) y
      p₁ x y x-rec = recB x
                (λ y' → P (x , y'))
                (λ y y-rec → f (x , y) (y-rec , x-rec))
                y

      p₂ : ∀ x → RecA (λ x' → ∀ y' → P (x' , y')) x
      p₂ = recA (λ x → ∀ y → P (x , y))
                (λ x x-rec y → f (x , y) (p₁ x y x-rec , x-rec))

      p₂x = p₂ x
```

We thus have:

```
Σ1-Rec Rec-A = Σ-Rec Rec-A λ _ → Rec-1

Σ1-builder rec-A = Σ-rec-builder rec-A (λ _ → rec-1-builder)
```

The `search` axtually exploited iterated lexicographic recursion on contexts, meaning that we can

- either take out a formula in bucket of order `n` and insert in any context of order `n`, or
- maintain the bucket size but act on a lower-order context

```
Rec-Context : (n : ℕ) → RecStruct (Context n)
Rec-Context zero = Rec-1
Rec-Context (suc n) = Σ-Rec Rec-Bucket λ _ → Rec-Context n

rec-Context-builder : ∀ {n} → RecursorBuilder (Rec-Context n)
rec-Context-builder {zero} = λ P x x₁ → tt
rec-Context-builder {suc n} = Σ-rec-builder rec-Bucket-builder (λ _ → rec-Context-builder {n})
```

Remark: These definition can be found (suitably generalized) in the Agda standard library:

```
open import Induction
open import Induction.Nat renaming (Rec to Rec-ℕ)
open import Induction.Lexicographic
```

2.5 Terminating search

We are left with translating our earlier definition, merely substituting recursion for pattern-matching, the type guiding us along the way:

```
⟨search[_]⟩ : {n : ℕ} (Γ : Context n) → Set
⟨search[ Γ ]⟩ = A → Bool

mutual
  search-step : ∀ {n} → (Γ : Context n) → Rec-Context n ⟨search[_]⟩ Γ → ⟨search[ Γ ]⟩
  search-step {zero} tt tt α = false
  search-step {suc n} ((zero , []) , Γ) (rec-Γ , tt) α =
    search-step  Γ rec-Γ α
  search-step {suc n}  ((suc l , B) , Γ) (rec-Γ , rec-B) α =
    let try = map (λ { (P , B) →  B / Γ [ P ]⊢ α }) (pick1 B) in
    any try ∨ search-step Γ rec-Γ α
    where _/_[_]⊢_ : Vec (Formula n) l → Context n → Formula n → A → Bool
          B / Γ [ Atom α       ]⊢ β = ⌊ α ≟ β ⌋
          B / Γ [ _⊃_ {n} P Q ]⊢ β  = B / Γ [ Q ]⊢ β ∧ B / Γ ⊢ P
            where  _/_⊢_ : Vec (Formula (suc n)) l → Context (suc n) → Formula n → Bool
                   B / Γ ⊢ Atom α = rec-B B Γ α
                   B / B₂ , Γ ⊢ P ⊃ Q  = B / B₂ , Γ ▹C P  ⊢ Q

  search : ∀ {n} →  (Γ : Context n) → ⟨search[ Γ ]⟩
  search {n} Γ = build (rec-Context-builder {n}) ⟨search[_]⟩ (search-step {n}) Γ



_⊢_ : ∀ {n} → Context n → Formula n → Bool
```

```
Γ ⊢ Atom α = search Γ α
Γ ⊢ P ⊃ Q  = Γ ▹C P  ⊢ Q

⊢_ : Formula 42 ⟶ Bool
⊢ P = []C ⊢ P
```

```
module TestMonotonic where

  open DjinnMonotonic ℕ Data.Nat._≟_

  A B ∐ : ∀ {n} ⟶ Formula n
  A = Atom 0
  B = Atom 1
  ∐ = Atom 2

  test₁ : ⊢ (A ⊃ B ⊃ A) ≡ true
  test₁ = refl

  test₂ : ⊢ (A ⊃ B) ≡ false
  test₂ = refl

  CPS : ∀ {n} ⟶ Formula n ⟶ Formula (2 + n)
  CPS A = (A ⊃ ∐) ⊃ ∐

  return : ⊢ (A ⊃ CPS A) ≡ true
  return =  refl

  bind : ⊢ (CPS A ⊃ (A ⊃ CPS B) ⊃ CPS B) ≡ true
  bind = refl

  call-cc : ⊢ (((A ⊃ CPS B) ⊃ CPS A) ⊃ CPS A) ≡ true
  call-cc = refl
```

CHAPTER
THREE

GENERAL RECURSION

Sometimes, we want to *write* a function and see later whether we want to *run* it totally (and, therefore, justify its termination in one way or another), or partially.

A more exhaustive presentation of the following ideas can be found in McBride's *Turing-Completeness Totally Free*.

3.1 Syntax for general recursion

We know of a good way to make (just) syntax: free term algebras! To describe a recursive function of type `(a : A) → B a`, we take the free monad of the signature `call : (a : A) → B a`:

```
data RecMon (X : Set) : Set where
  call : (a : A)(rec : B a → RecMon X) → RecMon X
  return : (x : X) → RecMon X
```

And its a monad:

```
monad : RawMonad RecMon
monad = record { return = return
                ; _>>=_ = _>>=_ }
       where  _>>=_ : ∀{X Y : Set} → RecMon X → (X → RecMon Y) → RecMon Y
              return x >>= f = f x
              call a rec >>= f = call a (λ b → (rec b) >>= f)
```

The operation *call* translates into the usual generic operation:

```
call⟨_⟩ : (a : A) → RecMon (B a)
call⟨ a ⟩ = call a return
```

Intuitively, the `call⟨_⟩` operation will be used as an oracle, providing a `B a` result to any `A` query. We thus write our recursive programs by calling the oracle instead of doing a recursive call.

We introduce some syntactic sugar to Pi-type the programs written in this syntax:

```
infix 2 Π-syntax

Π-syntax  : (A : Set)(B : A → Set) → Set
Π-syntax A B = (a : A) → RecMon (B a)
  where open RecMonad A B

syntax Π-syntax A (λ a → B) = Π[ a ∈ A ] B
```

Example: gcd We implement gcd pretty much as usual, using the oracle in the recursive cases:

```
gcd : Π[ mn ∈ ℕ × ℕ ] ℕ
gcd (0  , n)     = return n
gcd (m , 0)      = return m
gcd (suc m , suc n) with m ≤? n
... | yes _ = call⟨ suc m , n ∸ m ⟩
... | no  _ = call⟨ m ∸ n , suc n ⟩
```

Example: fib We can also chain recursive calls, as per the monadic structure. For example, we can write the naïve Fibonacci function:

```
fib : Π[ m ∈ ℕ ] ℕ
fib zero = return 0
fib (suc zero) = return 1
fib (suc (suc n)) = call⟨ suc n ⟩ >>= λ r₁ →
                    call⟨ n ⟩ >>= λ r₂ →
                    return (r₁ + r₂)
```

3.2 Monad morphism

In the following, we will implement a few interpretations of `RecMon` programs into some other monads. This begs the question: what does the monad morphisms from RecMon look like?

Let `M : Set → Set` be a monad. We have:

```
Monad(RecMon X, M X)
    ≅ Set(Σ[ a ∈ A ] B a → X, M X)    -- by the free/forgetful adjunction
    ≅ (a : A) → ∀ X → (B a → X) → M X -- by uncurry, etc.
    ≅ (a : A) → M (B a)               -- by Yoneda lemma
```

Or, put otherwise, a monad morphism from RecMon is entirely specified by a mere function of type `(a : A) → M (B a)`:

```
morph : ((a : A) → M (B a)) →
        ∀ {X} → RecMon X → M X
morph h (call a rec) = h a >>=-M λ b → morph h (rec b)
morph h (return x)   = return-M x
```

3.3 Interpretation: identity

There is a straightforward interpetation of `RecMon`, namely its interpretation into `RecMon`:

```
expand : Π[ a ∈ A ] B a → ∀ {X} → RecMon X → RecMon X
expand f = morph f
```

3.4 Interpretation: immediate values

We may blankly refuse to iterate:

```
already : ∀ {X} → RecMon X → Maybe X
already = morph (λ _ → nothing)
```

3.5 Interpretation: step-indexing

Iterating immediate interpretations, followed by the immediate one, we get a “step-indexed” interpretation:

```
engine : Π[ a ∈ A ] B a → ℕ → ∀ {X} → RecMon X → RecMon X
engine f zero = λ x → x
engine f (suc n) = engine f n ∘ expand f

petrol : Π[ a ∈ A ] B a → ℕ → (a : A) → Maybe (B a)
petrol f n = already ∘ engine f n ∘ f
```

This interpretation allows us to (maybe) run some programs:

```
test₁ : petrol fib 4 6 ≡ nothing
test₁ = refl

test₂ : petrol fib 5 6 ≡ just 8
test₂ = refl
```

3.6 Interlude: Universe of (collapsible) predicates

Coq users are familiar with the Prop universe, which is (essentially) a syntactic criteria for segregating computationally uninteresting objects (proofs) from the others (mostly, programs). Having identified such a fragment, we can erase it away at run-time.

There is no Prop in Agda. Instead, we adopt a semantic-based approach by defining a universe of inductive predicates in Agda and then prove that all its inhabitants are collapsible/proof-irrelevant. This terminology (and claim) will be formally justified in the last Section.

We thus define a set of *codes*:

```
data CDesc (I : Set) : Set₁ where
  `0 : CDesc I
  `1 : CDesc I
  `X : (i : I) → CDesc I
  _`×_ : (A B : CDesc I) → CDesc I
  `Π : (S : Set)(T : S → CDesc I) → CDesc I
```

Followed by their *interpretation*, which builds functors from `Set/I` to `Set`:

```
⟦_⟧ : {I : Set} → CDesc I → (I → Set) → Set
⟦ `0 ⟧ X = ⊥
⟦ `1 ⟧ X = ⊤
⟦ `X i ⟧ X = X i
⟦ A `× B ⟧ X = ⟦ A ⟧ X × ⟦ B ⟧ X
⟦ `Π S T ⟧ X = (s : S) → ⟦ T s ⟧ X
```

We obtain the code of (collapsible) descriptions, which describe endofunctors on `Set/I`:

```
record CFunc (I : Set) : Set₁ where
  constructor mk
  field
    func : I → CDesc I
```

From which we can define a generic least fixpoint operator, yielding the desired inductive predicates:

```
data μ {I : Set}(R : CFunc I)(i : I) : Set where
  con : ⟦ CFunc.func R i ⟧ (μ R) → μ R i
```

From there, we can also define induction over these structures, but we won't need it in this file. We will push this aspect further in the next lecture.

3.7 Collapsible accessibility predicate

From a function `f : Π[ a ∈ A ] B a`, we can build a Bove-Capretta predicate that, intuitively, is merely the reification (as an inductive predicate) of the call-graph of the recursive program.

As it turns out, this call-graph is always a collapsible predicate: to "prove" this, we simply describe it with a collapsible description:

```
dom : ∀{a} → RecMon (B a) → CDesc A
dom (return z) = `1
dom (call a rec) = `X a `× `Π (B a) λ b → dom (rec b)

Dom : CFunc A
Dom = CFunc.mk λ a → dom (f a)
```

Then, following the Bove-Capretta technique, we can run the (potentially general-recursive) function `f` by recursion over its call-graph (and, therefore, not over its arguments):

```
run : (a : A) → μ Dom a → B a
run1 : ∀{a} → (p : RecMon (B a)) → ⟦ dom p ⟧ B → B a
mapRun : ∀{a}{p : RecMon (B a)} → ⟦ dom p ⟧ (μ Dom) → ⟦ dom p ⟧ B

run a (con domS) = run1 (f a) (mapRun {p = f a} domS)

mapRun {p = return x} tt = tt
mapRun {p = call a rec} (domA , domRec) =
  run a domA , λ b → mapRun {p = rec b} (domRec b)

run1 (return b) tt = b
run1 (call a rec) (b , domRec) = run1 (rec b) (domRec b)
```

Note that we are *not* using the elements of μ `Dom s` in a computationally-relevant way: they are only here to convince Agda that the definition (trivially) terminates.

In fact, we know for sure that these elements cannot be computationally-relevant: being collapsible, there is nothing in μ `Dom` to compute with! At run-time, Inductive Families Need Not Store Their Indices and it can be entirely removed.

Example: gcd Applying our generic machinery to the recursive definition of gcd, we obtain the Bove-Capretta predicate:

```
DomGCD : ℕ × ℕ → Set
DomGCD (m , n) = μ (BC.Dom gcd) (m , n)
```

And, still applying our generic machinery, we get that, for any two input numbers satisfying the Bove-Capretta predicate, we can compute their gcd:

```
gcd-bove : (m n : ℕ) → DomGCD (m , n) → ℕ
gcd-bove m n xs = BC.run gcd (m , n) xs
```

Now, we can get rid of that pesky `DomGCD` predicate by proving, post facto, that our gcd function is indeed terminating. For that, we simply have to prove that `DomGCD` is inhabited for any input numbers m and n (the proof is not really important):

```
gcd-wf : (m n : ℕ) → DomGCD (m , n)
gcd-wf m n = build ([_⊗_] IndNat.<-rec-builder IndNat.<-rec-builder)
                 (λ { (m , n) → DomGCD (m , n) })
                 (λ { (m , n) rec → con (ih m n rec) })
                 (m , n)
       where ih : ∀ x y → (IndNat.<-Rec ⊗ IndNat.<-Rec) DomGCD (x , y) → ⟦ BC.dom gcd (gcd (x , y)) ⟧␣
↪DomGCD
             ih zero y rec = tt
             ih (suc x) zero rec = tt
             ih (suc x) (suc y) rec with x ≤? y
             ih (suc x) (suc y) (rec-x , rec-y) | yes p = (rec-x (y ∸ x) (s≤′s (≤⇒≤′ (n∸m≤n x y)))) , (λ _␣
↪→ tt)
             ih (suc x) (suc y) (rec-x , rec-y) | no ¬p = rec-y ((x ∸ y)) ((s≤′s (≤⇒≤′ (n∸m≤n y x)))) (suc y)␣
↪, (λ _ → tt)
```

And we get the desired gcd function:

```
gcd' : (m n : ℕ) → ℕ
gcd' m n = gcd-bove m n (gcd-wf m n)
```

3.8 Postlude: collapsible, formally

This is all very well but we've traded the freedom from termination checking for the burden of carrying Bove-Capretta witnesses around.

In Inductive Families Need Not Store Their Indices, Edwin Brady, Conor McBride, and James McKinna describe a *run-time* optimisation called "collapsing" (Section 6):

An inductive family `D : I → Set` is *collapsible* if

for every index i, if `a, b : D i`, then `a ≡ b` (extensionally)

That is, the index `i` determines entirely the content of the inductive family. Put otherwise, the inductive family has no computational content, hence the name "collapsible": morally, it collapses to a single element.

Remark: in the lingo of Homotopy Type Theory, a collapsible type `D : I → Set` corresponds to a family of h-propositions, ie. we have `∀ i → isProp(D i) ≜ ∀ i → ∀ (x y : D i) → x ≡ y`.

Example: relation (Section 6) Let us consider the comparison predicate:

```
data _≤`_ : ℕ → ℕ → Set where
  le0 : ∀{n} → 0 ≤` n
  leS : ∀{m n} → m ≤` n → suc m ≤` suc n
```

This datatype is collapsible:

```
≤-collapsible : ∀{m n} → (a b : m ≤` n) → a ≡ b
≤-collapsible {zero} le0 le0 = refl
≤-collapsible {suc m} {zero} () b
≤-collapsible {suc m} {suc n} (leS a) (leS b) rewrite ≤-collapsible a b = refl
```

Application: Assuming extensionality, we can prove (generically) that fixpoints of CDesc are indeed collapsible:

```
CDesc-collapse : ∀{I i}{R} → (xs ys : μ R i) → xs ≡ ys
CDesc-collapse {I}{R = R} (con xs) (con ys) = cong con (help (CFunc.func R _) xs ys)
  where postulate
          extensionality : {A : Set}{B : A → Set}{f g : (a : A) → B a} →
                           ((x : A) → (f x ≡ g x)) → f ≡ g

        help : (D : CDesc I) → (xs ys : ⟦ D ⟧ (μ R)) → xs ≡ ys
        help `0 () _
        help `1 tt tt = refl
        help (`X i) (con xs₁) (con ys₁) = cong con (help (CFunc.func R i) xs₁ ys₁)
        help (D₁ `× D₂) (xs₁ , xs₂) (ys₁ , ys₂) = cong₂ _,_ (help D₁ xs₁ ys₁) (help D₂ xs₂ ys₂)
        help (`Π S T) f g = extensionality λ s → help (T s) (f s) (g s)
```

Edwin's compiler should therefore be able to optimise away our Bove-Capretta predicates away (at run-time only!).