TYPES 2023

Composable partial functions in Coq, totally for free

Théo Winterhalter

Partial functions

Division

```
div : ℕ → ℕ → ℕ
div 0 m := 0
div n m := S (div (n - m) m)    (when n ≠ 0)
```

Partial functions

Division

```
div : ℕ → ℕ → ℕ
div 0 m := 0
div n m := S (div (n - m) m)    (when n ≠ 0)
```

Partial functions

Division

```
div : ℕ → ℕ → ℕ
div 0 m := 0
div n m := S (div (n - m) m)    (when n ≠ 0)
```

not even changing when m = 0

Partial functions

using fuel

```
div : ℕ → ℕ → ℕ → Fueled ℕ
```

Partial functions

using fuel

```
div : ℕ → ℕ → ℕ → Fueled ℕ
div 0 n m := NotEnoughFuel
```

Partial functions

using fuel

```
div : ℕ → ℕ → ℕ → Fueled ℕ
div 0 n m := NotEnoughFuel
div (S fuel) 0 m := ret 0
```

Partial functions

using fuel

```
div : ℕ → ℕ → ℕ → Fueled ℕ
div 0 n m := NotEnoughFuel
div (S fuel) 0 m := ret 0
div (S fuel) n m := q ← div (n - m) m ;; ret (S q)
```

Partial functions

using fuel

```
div : ℕ → ℕ → ℕ → Fueled ℕ
div 0 n m := NotEnoughFuel
div (S fuel) 0 m := ret 0
div (S fuel) n m := q ← div (n - m) m ;; ret (S q)
```

Problem: it remains after extraction!

Partial functions

Graphs

```
Inductive div_graph : ℕ → ℕ → ℕ → Prop :=
```

Partial functions

Graphs

```
Inductive div_graph : ℕ → ℕ → ℕ → Prop :=
| div_0 m : div_graph 0 m 0
```

Partial functions

Graphs

```
Inductive div_graph : ℕ → ℕ → ℕ → Prop :=
| div_0 m : div_graph 0 m 0
| div_rec n m q :
```

Partial functions

Graphs

```
Inductive div_graph : ℕ → ℕ → ℕ → Prop :=
| div_0 m : div_graph 0 m 0
| div_rec n m q :
    n ≠ 0 →
```

Partial functions

Graphs

```
Inductive div_graph : ℕ → ℕ → ℕ → Prop :=
| div_0 m : div_graph 0 m 0
| div_rec n m q :
    n ≠ 0 →
    div_graph (n - m) m q →
```

Partial functions

Graphs

```
Inductive div_graph : ℕ → ℕ → ℕ → Prop :=
| div_0 m : div_graph 0 m 0
| div_rec n m q :
    n ≠ 0 →
    div_graph (n - m) m q →
    div_graph n m (S q)
```

Partial functions

Graphs

```
Inductive div_graph : ℕ → ℕ → ℕ → Prop :=
| div_0 m : div_graph 0 m 0
| div_rec n m q :
    n ≠ 0 →
    div_graph (n - m) m q →
    div_graph n m (S q)
```

Doesn't even extract!

Partial functions

The Braga Method (Larchey-Wendling and Monin)

```
Inductive div_domain : ℕ → ℕ → Prop :=
| div_dom_0 m : div_domain 0 m
| div_dom_rec n m q :
    n ≠ 0 →
    div_graph (n - m) m q →
    div_domain n m
```

Definition by well-founded induction on the domain
which goes away at extraction

Contribution

A Coq library to do it all automatically

You write the following

```
Equations div : ∇ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

Contribution

A Coq library to do it all automatically

You write the following

```
Equations div : ∇ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

You get for free…

… a fueled version `fueled div`

Contribution

A Coq library to do it all automatically

You write the following

```
Equations div : ∇ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

You get for free…

… a fueled version `fueled div`

… a well-founded version `def div` (~ the Braga method)

Contribution

A Coq library to do it all automatically

You write the following

```
Equations div : ∇ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

and you get

```
Definition div_10_5 := div @ (10, 5).
```

Contribution

A Coq library to do it all automatically

You write the following

```
Equations div : ∇ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

and you get

```
Definition div_10_5 := div @ (10, 5).
Compute div_10_5. (* 2 *)
```

Contribution

A Coq library to do it all automatically

You write the following

```
Equations div : ∀ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

and you get

```
Definition div_10_5 := div @ (10, 5).
Compute div_10_5. (* 2 *)
Definition div_10_0 := div @ (10, 0). (* loops *)
```

General recursion monad

Following McBride's *Turing-completeness totally free*

```
Inductive orec A (B : A → Type) C :=
| _ret (x : C)
| _rec (x : A) (κ : B x → orec A B C)
| undefined.
```

General recursion monad

Following McBride's *Turing-completeness totally free*

arguments to recursive calls

```
Inductive orec A (B : A → Type) C :=
| _ret (x : C)
| _rec (x : A) (κ : B x → orec A B C)
| undefined.
```

General recursion monad

Following McBride's *Turing-completeness totally free*

arguments to recursive calls

(dependent)
results of recursive calls

```
Inductive orec A (B : A → Type) C :=
| _ret (x : C)
| _rec (x : A) (κ : B x → orec A B C)
| undefined.
```

General recursion monad

Following McBride's *Turing-completeness totally free*

arguments to recursive calls

(dependent)
results of recursive calls

return value

```
Inductive orec A (B : A → Type) C :=
| _ret (x : C)
| _rec (x : A) (κ : B x → orec A B C)
| undefined.
```

General recursion monad

Following McBride's *Turing-completeness totally free*

arguments to recursive calls

(dependent)
results of recursive calls

return value

```
Inductive orec A (B : A → Type) C :=
| _ret (x : C)
| _rec (x : A) (κ : B x → orec A B C)
| undefined.
```

```
∇ (x : A), B := ∀ x, orec A (λ x, B) B
```

Semantics through the graph

Given `f : ∇ (x : A), B x`

Semantics through the graph

Given `f : ∀ (x : A), B x`

```
Inductive orec_graph {a} : orec A B (B a) → B a → Prop :=
```

Semantics through the graph

Given f `: ∀ (x : A), B x`

```
Inductive orec_graph {a} : orec A B (B a) → B a → Prop :=
```

```
―――――――――――――――――――――――――――
orec_graph (_ret x) x
```

Semantics through the graph

Given f : ∇ (x : A), B x

```
Inductive orec_graph {a} : orec A B (B a) → B a → Prop :=
```

```
────────────────────────────
orec_graph (_ret x) x
```

```
  orec_graph (f x) w
  orec_graph (κ v) w
────────────────────────────
orec_graph (_rec x κ) w
```

Semantics through the graph

Given f `: ∇ (x : A), B x`

```
Inductive orec_graph {a} : orec A B (B a) → B a → Prop :=
```

```
─────────────────────────
orec_graph (_ret x) x
```

```
   orec_graph (f x) w
   orec_graph (κ v) w
──────────────────────────
orec_graph (_rec x κ) w
```

```
Definition graph f x v := orec_graph (f x) v.
```

Semantics through the graph

Given `f : ∇ (x : A), B x`

```
Inductive orec_graph {a} : orec A B (B a) → B a → Prop :=
```

```
──────────────────────────
orec_graph (_ret x) x
```

```
   orec_graph (f x) w
   orec_graph (κ v) w
──────────────────────────
orec_graph (_rec x κ) w
```

```
Definition graph f x v := orec_graph (f x) v.
```

```
Definition domain f x := ∃ v, graph f x v.
```

Instances

Given `f : ∇ (x : A), B x` we get

```
fueled f : ℕ → ∀ (x : A), Fueled (B x)
```

```
def f : ∀ (x : A), domain f x → B x
```

Instances

Given `f : ∀ (x : A), B x` we get

```
fueled f : ℕ → ∀ (x : A), Fueled (B x)
```

```
def f : ∀ (x : A), domain f x → B x
```

Exponential instances so they can run virtually forever

Instances

Given `f : ∇ (x : A), B x` we get

```
fueled f : ℕ → ∀ (x : A), Fueled (B x)
```

```
def f : ∀ (x : A), domain f x → B x
```

Exponential instances so they can run virtually forever

`f @ u` is actually `def f u _`
where the domain proof is inferred by running the fueled version

Instances

Given `f : ∇ (x : A), B x` we get

```
fueled f : ℕ → ∀ (x : A), Fueled (B x)
```

```
def f : ∀ (x : A), domain f x → B x
```

Exponential instances so they can run virtually forever

`f @ u` is actually `def f u _`
where the domain proof is inferred by running the fueled version

We show they respect the graph
and give reasoning principles (functional induction, domain proofs) on them

Composing partial functions

Directly in the source language

```
Equations div : ∇ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

```
Equations test_div : ∇ (p : ℕ × ℕ), bool :=
  test_div (n, m) := q ← call div (n, m) ;; ret (q * m =? n).
```

Composing partial functions

Directly in the source language

```
Equations div : ∇ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

```
Equations test_div : ∇ (p : ℕ × ℕ), bool :=
  test_div (n, m) := q ← call div (n, m) ;; ret (q * m =? n).
```

we don't want to inline functions!

Composing partial functions

Directly in the source language

```
Equations div : ∇ (p : ℕ × ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n - m, m) ;; ret (S q).
```

```
Equations test_div : ∇ (p : ℕ × ℕ), bool :=
  test_div (n, m) := q ← call div (n, m) ;; ret (q * m =? n).
```

we don't want to inline functions!

and anyway we cannot always do it

Composing partial functions

We extend the free monad

```
Inductive orec A (B : A → Type) C :=
| _ret (x : C)
| _rec (x : A) (κ : B x → orec A B C)
| _call f `{PFun f} (x : src f) (κ : tgt f x → orec A B C)
| undefined.
```

Composing partial functions

We extend the free monad

```
Inductive orec A (B : A → Type) C :=
| _ret (x : C)
| _rec (x : A) (κ : B x → orec A B C)
| _call f `{PFun f} (x : src f) (κ : tgt f x → orec A B C)
| undefined.
```

type class of objects with graph, domain,
fuel and wf versions…

Composing partial functions

We extend the free monad

```
Inductive orec A (B : A → Type) C :=
| _ret (x : C)
| _rec (x : A) (κ : B x → orec A B C)
| _call f `{PFun f} (x : src f) (κ : tgt f x → orec A B C)
| undefined.
```

type class of objects with graph, domain, fuel and wf versions…

We give an instance for every `f : ∀ (x : A), B x`

```
Equations div : ∇ (p : nat * nat), nat :=
  div (0, m) := ret 0 ;
  div (n, m) := S <*> rec (n - m, m).

Equations test_div : ∇ (p : nat * nat), bool :=
  test_div (n, m) := q ← call div (n, m) ;; ret (q * m =? n).

Definition div_10_5 := div @ (10, 5).
Fail Timeout 1 Definition div_10_0 := div @ (10, 0).

Compute div @ (50,6).

```

PROBLEMS TERMINAL OUTPUT DEBUG CONSOLE GITLENS

```
The command has indeed failed with message:
Timeout!
     = 9
     : nat
```

```
Lemma div_domain :
  ∀ n m,
    (n = 0 ∨ m ≠ 0) →
    domain div (n, m).
Proof.
  intros n m hm.
  assert (hw : WellFounded lt).
  { exact _. }
  specialize (hw n). induction hw as [n hn ih].
  apply compute_domain. funelim (div (n, m)). all: cbn - ["-"].
  - constructor.
  - split. 2: auto.
    apply ih. all: lia.
Qed.
```

```
m, n : nat
hm : n = 0 ∨ m ≠ 0
hn : ∀ y : nat, y < n → Acc lt y
ih : ∀ y : nat, y < n → y = 0 ∨ m ≠ 0 → domain div (y, m)
______________________________________
(1/1)
domain div (n, m)
```

```
Lemma div_domain :
  ∀ n m,
    (n = 0 ∨ m ≠ 0) →
    domain div (n, m).
Proof.
  intros n m hm.
  assert (hw : WellFounded lt).
  { exact _. }
  specialize (hw n). induction hw as [n hn ih].
  apply compute_domain. funelim (div (n, m)). all: cbn - ["-"].
  - constructor.
  - split. 2: auto.
    apply ih. all: lia.
Qed.
```

```
m :  nat
hm :  0 = 0 ∨ m ≠ 0
hn :  ∀ y : nat, y < 0 → Acc lt y
ih :  ∀ y : nat, y < 0 → y = 0 ∨ m ≠ 0 → domain div (y, m)
________________________________________
(1/2)
True
________________________________________
(2/2)
domain div (S n - m, m)
∧ (∀ v : nat, graph div (S n - m, m) v → True)
```

Lemma div_domain_implies :
∀ n m,
domain div (n, m) →
n = 0 ∨ m ≠ 0.
Proof.
assert (h : funind div (λ _, True) (λ '(n, m) _, n = 0 ∨ m ≠ 0)).
{ intros [n m] _.
funelim (div (n, m)). all: cbn - ["-"].
- left. reflexivity.
- intuition lia.
}
intros n m [v hd].
funind h in hd. assumption.
Qed.
m : nat
(1/2)
0 = 0 ∨ m ≠ 0
(2/2)
True ∧ (nat → S n - m = 0 ∨ m ≠ 0 → S n = 0 ∨ m ≠ 0)

```
Lemma div_domain_implies :
  ∀ n m,
    domain div (n, m) →
    n = 0 ∨ m ≠ 0.
Proof.
  assert (h : funind div (λ _, True) (λ '(n, m) _, n = 0 ∨ m ≠ 0)).
  { intros [n m] _.
    funelim (div (n, m)). all: cbn - ["-"].
    - left. reflexivity.
    - intuition lia.
  }
  intros n m [v hd].
  funind h in hd. assumption.
Qed.
```

```
h :  funind div (λ _ : nat * nat, True) (λ '(n, m) (_ : nat), n =
n, m, v :  nat
hd :  n = 0 ∨ m ≠ 0
________________________________________
(1/1)
n = 0 ∨ m ≠ 0
```

Perspectives

Support for effects, more general recursion monad (for universe issues)

```
Equations div : ∇ (p : ℕ × ℕ), exn error ♯ ℕ :=
  div (n, 0) := raise DivisionByZero ;
  div (0, m) := ret 0 ;
  div (n, m) := S · rec (n - m, m).
```

Thanks: Jason Gross, Meven Lennon-Bertrand, Kenji Maillard

Applications: logical relations for MLTT (Adjedj *et al.*), MetaCoq with rewrite rules

/TheoWinterhalter/coq-partialfun