Formalizing Computability Theory via Partial Recursive Functions

Mirja van de Pol - s1061675 - MFoCS
Robin Holen - s1017668 - MFoCS

Radboud University

December 20, 2021

Introduction

- Paper:
  Mario Carneiro, Formalizing Computability Theory via Partial Recursive Functions
- Small overview:
  - Goal: making an extension to mathlib library of lean, formalizing *computability theory*.
  - We will encode computable functions using the definition with Kleene's recursive functions
    - Primitive recursive functions
    - Partial/mu recursive functions
    - Example of an important theorem

Numbering

When is a type *encodable*?

```
1  class encodable (α : Type u) :=
2      (encode : α → nat)
3      (decode : nat → option α)
4      (encode : forall a, decode (encode a) = some a)
```

Numbering

When is a type *encodable*?

```
class encodable (α : Type u) :=
    (encode : α → nat)
    (decode : nat → option α)
    (encode : forall a, decode (encode a) = some a)
```

Some examples of an encoding:

```
variables {α β} [encodable α] [encodable β]
def encode_sum : α ⊕ β → N
| (inl a) := 2 * encode a
| (inr b) := 2 * encode b + 1
def encode_prod : α × β → N
| (a, b) := mkpair (encode a) (encode b)
def encode_option : option α → N
| none := 0
| (some a) := succ (encode a)
```

Primitive recursive functions

- $n \mapsto 0$ is primitive recursive.
- $n \mapsto S(n)$ is primitive recursive.
- $(n_0, \ldots, n_{k-1}) = n_i$ is primitive recursive.
- If $f : \mathbb{N}^k \to \mathbb{N}$ and $g_i : \mathbb{N}^m \to \mathbb{N}$ are primitive recursive then $v \mapsto f(g_0(v), \ldots, g_{k-1}(v))$ is primitive recursive.
- If $f : \mathbb{N}^m \to \mathbb{N}$ and $g : \mathbb{N}^{m+2} \to \mathbb{N}$ are primitive recursive, then the function $h : \mathbb{N}^{m+1} \to \mathbb{N}$ defined by:

$$h(\vec{z}, 0) = f(\vec{z})$$
$$h(\vec{z}, S(n)) = g(\vec{z}, n, h(\vec{z}, n))$$

  is primitive recursive

Primitive recursive functions

```
1  inductive primrec : (ℕ → ℕ) → Prop
```

Primitive recursive functions

```
1  inductive primrec : (ℕ → ℕ) → Prop
2      | zero : primrec (λ n, 0)
3      | succ : primrec succ
```

Primitive recursive functions

```
1  inductive primrec : (ℕ → ℕ) → Prop
2      | zero : primrec (λ n, 0)
3      | succ : primrec succ
4      | left : primrec (λ n, fst (unpair n))
5      | right : primrec (λ n, snd (unpair n))
```

Primitive recursive functions

```
1  inductive primrec : (ℕ → ℕ) → Prop
2      | zero : primrec (λ n, 0)
3      | succ : primrec succ
4      | left : primrec (λ n, fst (unpair n))
5      | right : primrec (λ n, snd (unpair n))
6      | pair {f g} : primrec f → primrec g →
7        primrec (λ n, mkpair (f n) (g n))
8      | comp {f g} : primrec f → primrec g →
9        primrec (f · g)
```

Primitive recursive functions

```
1  inductive primrec : (ℕ → ℕ) → Prop
2      | zero : primrec (λ n, 0)
3      | succ : primrec succ
4      | left : primrec (λ n, fst (unpair n))
5      | right : primrec (λ n, snd (unpair n))
6      | pair {f g} : primrec f → primrec g →
7        primrec (λ n, mkpair (f n) (g n))
8      | comp {f g} : primrec f → primrec g →
9        primrec (f · g)
10     | prec {f g} : primrec f → primrec g →
11       primrec (unpaired (λ z n, nat.rec_on n (f z)
12         (λ y IH, g (mkpair z (mkpair y IH)))))
```

Primitive recursive functions

Note: only `primrec` only defined for functions $\mathbb{N} \to \mathbb{N}$. So we will extend for functions on other types.

- We use `encodable` to encode any instance of some type to $\mathbb{N}$ and back.
- Problem: `encode` $\circ$ `decode` might not be primitive recursive.
- Introducing a new class: `Primcodable`. An instance of `encodable` where `encode` $\circ$ `decode` is primitive recursive, is called: `primcodable`.

This gives definition:

```
1 def primrec {α, β} [primcodable α] [primcodable β]
2   (f : α → β) : Prop :=
3   nat.primrec (λ n, encode (option.map f (decode α n)))
```

Primitive recursive functions

- Noteworthy is that `list` α is `primcodable`
- This due to the fact that `fold:`$(\alpha \to \beta \to \alpha) \to \alpha \to$ `list` $\beta \to \alpha$ is `primcodable`.
- Now the following theorem could be proven when we allow `list` α to be an input:

```
theorem nat_strong_rec
(f : α → ℕ → σ )
{g : α → list σ → option σ}
(hg : primrec₂ g)
(H : ∀ a n, g a (map (f a) (range n)) = some (f a n)) :
primrec₂ f
```

The μ-recursive functions

Definition of μ-recursive functions

- $n \mapsto 0$ is μ-recursive.
- $n \mapsto S(n)$ is μ-recursive.
- $(n_0, \ldots, n_{k-1}) = n_i$ is μ-recursive.
- If $f : \mathbb{N}^k \to \mathbb{N}$ and $g_i : \mathbb{N}^m \to \mathbb{N}$ are μ-recursive then $v \mapsto f(g_0(v), \ldots, g_{k-1}(v))$ is μ-recursive.
- If $f : \mathbb{N}^m \to \mathbb{N}$ and $g : \mathbb{N}^{m+2} \to \mathbb{N}$ are μ-recursive, then the function $h : \mathbb{N}^{m+1} \to \mathbb{N}$ defined by:

$$h(\vec{z}, 0) = f(\vec{z})$$
$$h(\vec{z}, S(n)) = g(\vec{z}, n, h(\vec{z}, n))$$

is μ-recursive.

The μ-recursive functions

Definition of μ-recursive functions

- $n \mapsto 0$ is μ-recursive.
- $n \mapsto S(n)$ is μ-recursive.
- $(n_0, \ldots, n_{k-1}) = n_i$ is μ-recursive.
- If $f : \mathbb{N}^k \to \mathbb{N}$ and $g_i : \mathbb{N}^m \to \mathbb{N}$ are μ-recursive then $v \mapsto f(g_0(v), \ldots, g_{k-1}(v))$ is μ-recursive.
- If $f : \mathbb{N}^m \to \mathbb{N}$ and $g : \mathbb{N}^{m+2} \to \mathbb{N}$ are μ-recursive, then the function $h : \mathbb{N}^{m+1} \to \mathbb{N}$ defined by:

$$h(\vec{z}, 0) = f(\vec{z})$$
$$h(\vec{z}, S(n)) = g(\vec{z}, n, h(\vec{z}, n))$$

  is μ-recursive.
- If $p : \mathbb{N}^{k+1} \to$ `Bool` is a total μ-recursive predicate, then $f(\vec{m}) = \mu\, n[p(n, \vec{m})]$, which computes the smallest n such that $p(n, \vec{m})$ is `true`, is μ-recursive.

The μ-recursive functions

μ-recursive functions are in general not total functions.

The μ-recursive functions

μ-recursive functions are in general not total functions.

An example:

If $p(n,z) = \begin{cases} \texttt{true} & \texttt{if } n^2 = z \\ \texttt{false} & \texttt{if } n^2 \neq z \end{cases}$

Then $\mu\, n[p(n,5)]$ will never halt and so it is not defined.

The μ-recursive functions

- To represent nontermination, we can use type `part` α:

```
1  def part (α : Type*) := Σ p : Prop, (p → α)
```

The μ-recursive functions

- To represent nontermination, we can use type `part` α:

```
1  def part (α : Type*) := Σ p : Prop, (p → α)
```

- Here `p:part` α, `p`$=\langle p_1, p_2 \rangle$ is a dependant pair of a proposition p_1 and a function $p_2 : p_1 \to \alpha$ from **proofs** of p_1 to α. Here $\bot = \langle$`false, exfalso`$\rangle$

The μ-recursive functions

- To represent nontermination, we can use type `part` α:

```
1 def part (α : Type*) := Σ p : Prop, (p → α)
```

- Here `p:part` α, `p`$=\langle p_1, p_2 \rangle$ is a dependant pair of a proposition p_1 and a function $p_2 : p_1 \to \alpha$ from **proofs** of p_1 to α. Here $\bot = \langle$`false, exfalso`$\rangle$
- We can not decide if `part` α contains a value but if it does, we can extract it by using p_2 and apply it to p_1.

The μ-recursive functions

- To represent nontermination, we can use type `part` α:

```
1 def part (α : Type*) := Σ p : Prop, (p → α)
```

- Here `p:part` α, `p`$=\langle p_1, p_2 \rangle$ is a dependant pair of a proposition p_1 and a function $p_2 : p_1 \to \alpha$ from **proofs** of p_1 to α. Here $\bot = \langle$`false, exfalso`$\rangle$
- We can not decide if `part` α contains a value but if it does, we can extract it by using p_2 and apply it to p_1.
- `part` α has two functions that come with it:
  - `pure:` $\alpha \to$ `part` α
  - `pure a` $= \langle$`true`$, \lambda\_.a \rangle$
  - `bind:` `part` $\alpha \to (\alpha \to$ `part` $\beta) \to$ `part` β
  - `bind` $\langle q, f \rangle g = \langle (\exists\, h : q[(g\ (f\ h))_1]), (\lambda\, h.(g\ (f\ h_1)_2)h_2) \rangle$

The μ-recursive functions

`bind` has an infix operator $\gg=$

- `bind` $p\ g = p \gg= g$ where $p = \langle q, f \rangle$

If we define $a \in p$ as "p is defined and is equal to a", then we can express a property of `bind` easily as

- $b \in (p \gg= f) \leftrightarrow \exists\, a \in p[b \in f\ a]$

The μ-recursive functions

`bind` has an infix operator $\gg=$

- `bind` $p\ g = p \gg= g$ where $p = \langle q, f \rangle$

If we define $a \in p$ as "p is defined and is equal to a", then we can express a property of `bind` easily as

- $b \in (p \gg= f) \leftrightarrow \exists\, a \in p[b \in f\ a]$

In addition to `pure` and `bind`, `part` α also comes with `map` also with an infix operator <$>.
Here $\alpha \nrightarrow \beta$ is the same as $\alpha \to$ `part` β

- `map`$:(\alpha \to \beta) \nrightarrow \alpha \nrightarrow \beta$
- f<$>$p$ = \langle p_1, f \circ p_2 \rangle$

The μ-recursive functions

Properties of `fix`

1. `fix (f :` $\alpha \nrightarrow \beta \oplus \alpha) : \alpha \nrightarrow \beta$
2. `b` $\in$ `fix f a` $\leftrightarrow$ `inl b` $\in$ `f a` $\vee$
3. $\exists$ `a', inr a'` $\in$ `f a` $\wedge$ `b` $\in$ `fix f a'`

The μ-recursive functions

Properties of `fix`

1 `fix (f :` $\alpha \nrightarrow \beta \oplus \alpha) : \alpha \nrightarrow \beta$

2 `b` $\in$ `fix f a` $\leftrightarrow$ `inl b` $\in$ `f a` $\vee$

3 $\exists$ `a', inr a'` $\in$ `f a` $\wedge$ `b` $\in$ `fix f a'`

What is the input of `fix`?

`fix` gets a function (f : $\alpha \nrightarrow \beta \oplus \alpha$) and an instance of type α.

The μ-recursive functions

Properties of `fix`

1 `fix (f :` $\alpha \nrightarrow \beta \oplus \alpha) : \alpha \nrightarrow \beta$

2 `b` $\in$ `fix f a` $\leftrightarrow$ `inl b` $\in$ `f a` $\vee$

3 $\exists$ `a', inr a'` $\in$ `f a` $\wedge$ `b` $\in$ `fix f a'`

What is the input of `fix`?

`fix` gets a function (f : $\alpha \nrightarrow \beta \oplus \alpha$) and an instance of type α.

What is the output of `fix`?

With the input *a*, `fix` computes *fa*. This gives three options:

- If f a $\in \alpha$, then `fix` recursively computes f^n a,

The μ-recursive functions

Properties of `fix`

1 `fix (f :` $\alpha \nrightarrow \beta \oplus \alpha) : \alpha \nrightarrow \beta$

2 `b` $\in$ `fix f a` $\leftrightarrow$ `inl b` $\in$ `f a` $\vee$

3 $\exists$ `a', inr a'` $\in$ `f a` $\wedge$ `b` $\in$ `fix f a'`

What is the input of `fix`?

`fix` gets a function (f : $\alpha \nrightarrow \beta \oplus \alpha$) and an instance of type α.

What is the output of `fix`?

With the input *a*, `fix` computes *fa*. This gives three options:

- If f a $\in \alpha$, then `fix` recursively computes f^n a,
- If f a $\in \beta$, then `fix` returns β,

The μ-recursive functions

Properties of `fix`

1 `fix (f :` $\alpha \nrightarrow \beta \oplus \alpha) : \alpha \nrightarrow \beta$

2 `b` $\in$ `fix f a` $\leftrightarrow$ `inl b` $\in$ `f a` $\vee$

3 $\exists$ `a', inr a'` $\in$ `f a` $\wedge$ `b` $\in$ `fix f a'`

What is the input of `fix`?

`fix` gets a function (f : $\alpha \nrightarrow \beta \oplus \alpha$) and an instance of type α.

What is the output of `fix`?

With the input *a*, `fix` computes *fa*. This gives three options:

- If f a $\in \alpha$, then `fix` recursively computes f^n a,
- If f a $\in \beta$, then `fix` returns β,
- If f a gives error, then `fix` returns an error.

The μ-recursive functions

Using the definition of `fix`, we can define `find`:

```
1  find : (ℕ ⇸ bool) ⇸ ℕ
2  find p = fix (λn. if p n then inl n else inr(n + 1)) 0
```

The μ-recursive functions

Using the definition of `fix`, we can define `find`:

```
1 find : (ℕ ⇸ bool) ⇸ ℕ
2 find p = fix (λn. if p n then inl n else inr(n + 1)) 0
```

This is exactly $\mu\, n[p(n)]$!

The μ-recursive functions

```
1  inductive partrec : (ℕ→.ℕ) → Prop
2  | zero : partrec (pure 0)
3  | succ : partrec succ
4  | left : partrec (λ n, fst (unpair n))
5  | right : partrec (λ n, snd (unpair n))
6  | pair {f g} : partrec f → partrec g →
7  partrec (λ n, f n >>= λ a, g n >>= λ b, pure (mkpair a b))
8  | comp {f g} : partrec f → partrec g →
9  partrec (λ n, g n >>= f)
10 | prec {f g} : partrec f → partrec g →
11 partrec (unpaired (λ a n, nat.rec_on n (f a)
12 (λ y IH, IH >>= λ i,
13 g (mkpair a (mkpair y i)))))
```

The μ-recursive functions

```
1  inductive partrec : (ℕ→.ℕ) → Prop
2  | zero : partrec (pure 0)
3  | succ : partrec succ
4  | left : partrec (λ n, fst (unpair n))
5  | right : partrec (λ n, snd (unpair n))
6  | pair {f g} : partrec f → partrec g →
7  partrec (λ n, f n >>= λ a, g n >>= λ b, pure (mkpair a b))
8  | comp {f g} : partrec f → partrec g →
9  partrec (λ n, g n >>= f)
10 | prec {f g} : partrec f → partrec g →
11 partrec (unpaired (λ a n, nat.rec_on n (f a)
12 (λ y IH, IH >>= λ i,
13 g (mkpair a (mkpair y i)))))
14 | find f : partrec f → partrec (λ a,
15 find (λ n, (λ m, m = 0) <$> f (mkpair a n)))
```

The μ-recursive functions

Definition: `(λ a, find (λ n, (λ m, m = 0) <$> f (mkpair a n)))`.

The μ-recursive functions

Definition: `(λ a, find (λ n, (λ m, m = 0) <$> f (mkpair a n)))`.

- `mkpair a n` is a natural number,

The μ-recursive functions

Definition: `(λ a, find (λ n, (λ m, m = 0) <$> f (mkpair a n)))`.

- `mkpair a n` is a natural number,
- `f mkpair a n` returns an instance of `part` $\mathbb{N}$,

The μ-recursive functions

Definition: `(λ a, find (λ n, (λ m, m = 0) <$> f (mkpair a n)))`.

- `mkpair a n` is a natural number,
- `f mkpair a n` returns an instance of `part` $\mathbb{N}$,
- `(λ m , m=0) <$> ( f mkpair a n)` is an instance of `part bool`,
  this is True is `f mkpair a n` 'equals' 0, and False if it 'is' a natural number, and 'undefined' if `f mkpair a n` is 'undefined'.

The μ-recursive functions

Definition: `(λ a, find (λ n, (λ m, m = 0) <$> f (mkpair a n)))`.

- `mkpair a n` is a natural number,
- `f mkpair a n` returns an instance of `part` $\mathbb{N}$,
- `(λ m , m=0) <$> ( f mkpair a n)` is an instance of `part bool`,
  this is True is `f mkpair a n` 'equals' 0, and False if it 'is' a natural number, and 'undefined' if `f mkpair a n` is 'undefined'.
- `find _` now has as input an instance of $\mathbb{N} \nrightarrow$`Bool`, and returns the smallest number where `f mkpair a n` equals 0.

The μ-recursive functions

Definition: `(λ a, find (λ n, (λ m, m = 0) <$> f (mkpair a n)))`.

- `mkpair a n` is a natural number,
- `f mkpair a n` returns an instance of `part` $\mathbb{N}$,
- `(λ m , m=0) <$> ( f mkpair a n)` is an instance of `part bool`,
  this is True is `f mkpair a n` 'equals' 0, and False if it 'is' a natural number, and 'undefined' if `f mkpair a n` is 'undefined'.
- `find _` now has as input an instance of $\mathbb{N} \nrightarrow$`Bool`, and returns the smallest number where `f mkpair a n` equals 0.
- The μ-function is the function that seeks for some a the smallest n where `f mkpair a n` equals zero. Either it gives an n, or it is undefined. Their is no direct way of getting the answer, because the answer is wrapped within `part` $\mathbb{N}$.

Code

Inductive representation of the syntax of partial recursive functions

```
1  inductive code : Type
2  | zero : code
3  | succ : code
4  | left : code
5  | right : code
6  | pair : code → code → code
7  | comp : code → code → code
8  | prec : code → code → code
9  | find' : code → code
```

Code evaluation

Evaluating Code can be done quite easily by the function `eval`

```
def eval : code → ℕ →. ℕ
| zero := pure 0
| succ := succ
| left := λ n, n.unpair.1
| right := λ n, n.unpair.2
| (pair cf cg) := λ n,
eval cf n >>= λ a, eval cg n >>= λ b, pure (mkpair a b)
| (comp cf cg) := λ n, eval cg n >>= eval cf
| (prec cf cg) := unpaired (λ a n,
nat.rec_on n (eval cf a) (λ y IH, IH >>= λ i,
eval cg (mkpair a (mkpair y i))))
| (find' cf) := unpaired (λ a m, (λ i, i + m) <$>
find (λ n, (λ m, m = 0) <$> eval cf (mkpair a (n + m))))
```

Halting Problem

The halting problem states:
The set $\{c : \text{code} \mid \text{eval } c\ 0 \text{ is defined}\}$ is not computable.
This can now be proven in Lean.

Conclusion

- We formalized primitive recursive functions in Lean for any `primcodable` type α
- Furthermore, we formalized partial recursive functions by introducing a partiality monad to wrap a type α into `part` α and by defining the minimisation operator in terms of `fix`
- We created `code` and `eval` to prove many important theorems within computability theory.

Merry Christmas!