Formalizing computability theory via partial recursive functions

Mario Carneiro

Carnegie Mellon University

September 9, 2019

Lean

- An open source interactive theorem prover developed primarily by Leonardo de Moura (Microsoft Research)
- Focus on software verification and formalized mathematics
- Based on Dependent Type Theory
  - Classical, non-HoTT
  - Similar to CIC, the axiom system used by Coq
- Lean 3 includes a powerful metaprogramming infrastructure for Lean in Lean
- The mathlib library for Lean 3 provides a broad range of pure mathematics and tools for (meta)programming
  - Includes abstract algebra, category theory, **computability theory**, and much much more. . .
- This formalization is available online[1]

[1] `https://github.com/leanprover-community/mathlib/tree/master/src/computability`

How to start?

- Computability theory has multiple “competing” formalizations
  - Turing machines
  - Lambda calculus
  - Partial recursive functions
  - Minsky register machines
  - Wang B-machines
  - …

How to start?

- Computability theory has multiple “competing” formalizations
  - Turing machines
  - Lambda calculus
  - Partial recursive functions
  - Minsky register machines
  - Wang B-machines
  - …
- …and they are all equivalent (Church-Turing thesis)

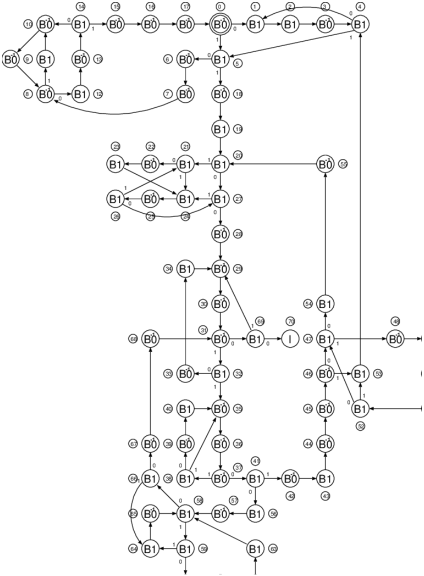
How to start?

- Computability theory has multiple “competing” formalizations
  - Turing machines
  - Lambda calculus
  - Partial recursive functions
  - Minsky register machines
  - Wang B-machines
  - ...
- ... and they are all equivalent (Church-Turing thesis)

In theory it shouldn’t matter, but in practice different ways of saying the same thing make a *huge* difference in the difficulty of a formalization project. The only way to get good data is to try all the ways and see which is easiest.

Turing machines

- Very simple to describe, very hard to use
- Mathematical description bears obvious resemblance to von Neumann architecture and physical machines of Turing's day
- Non-compositional semantics and graph based transitions make them unnatural in a formal proof



Post-Turing machines / Wang B-machines

- Various variations and limitations of Turing machines make them more like standard processors
  - not every instruction is a jump
- By limiting control flow patterns, this can be made compositional
- Memory access is still non-compositional

```
write 0
write 1
move left
move right
if read 0 then goto i
if read 1 then goto j
```

Post-Turing machines / Wang B-machines

- Various variations and limitations of Turing machines make them more like standard processors
  - not every instruction is a jump
- By limiting control flow patterns, this can be made compositional
- Memory access is still non-compositional

```
write 0
write 1
move left
move right
if read 0 then goto i
if read 1 then goto j
```

Verdict: This is a good way to wrangle Turing machines, but it is still too concrete; proofs involve many details of the model and the ideas are lost in the noise. (Concrete is useful, but concrete + unrealistic is not.)

Lambda calculus

$$e ::= x \mid e\ e' \mid \lambda x.\, e$$

$$(\lambda x.\, e)\ a \longrightarrow e[a/x]$$

- Simple, abstract, and compositional
- Partiality and confluence are new problems in this setting
  - Turing machines aren't compositional so there is only one place that partiality can arise, but any subterm of a lambda term can fail to terminate.
  - Confluence can be addressed by picking an evaluation order, or by proving Church-Rosser.
- Lean is a dependent type theory so it is itself a variant on lambda calculus, but we can't reuse this lambda as a lambda term (a.k.a. higher order abstract syntax (HOAS)).
- Lean needs all terms to terminate, so a direct embedding is difficult

Lambda calculus

$$e ::= x \mid e\ e' \mid \lambda x.\, e$$

$$(\lambda x.\, e)\ a \longrightarrow e[a/x]$$

- Simple, abstract, and compositional
- Partiality and confluence are new problems in this setting
  - Turing machines aren't compositional so there is only one place that partiality can arise, but any subterm of a lambda term can fail to terminate.
  - Confluence can be addressed by picking an evaluation order, or by proving Church-Rosser.
- Lean is a dependent type theory so it is itself a variant on lambda calculus, but we can't reuse this lambda as a lambda term (a.k.a. higher order abstract syntax (HOAS)).
- Lean needs all terms to terminate, so a direct embedding is difficult
- Verdict: weak reject

Primitive recursive functions

```
inductive primrec : (ℕ → ℕ) → Prop
| zero : primrec (λ n, 0)
| succ : primrec succ
| left : primrec (λ n, fst (unpair n))
| right : primrec (λ n, snd (unpair n))
| pair {f g} : primrec f → primrec g →
  primrec (λ n, mkpair (f n) (g n))
| comp {f g} : primrec f → primrec g →
  primrec (f ∘ g)
| prec {f g} : primrec f → primrec g →
  primrec (unpaired (λ z n, nat.rec_on n (f z)
     (λ y IH, g (mkpair z (mkpair y IH)))))
```

Primitive recursion asserts that if f and g are primrec then so is the function h such that

$$h(z,0) = f(z) \quad \text{and} \quad h(z,n+1) = g(z,(n,h(z,n)))$$

Primitive recursive functions

- Primitive recursive functions are *functions* in Lean's logic
- All primrec functions terminate, so Lean can evaluate them without issue
- No n-ary functions needed because we embed projection and pairing relative to Cantor's pairing function
  - composition is simply f, g primrec $\rightarrow f \circ g$ primrec
- Although slightly more complicated to define, primrec functions come with enough "built in" for us to get addition and multiplication very easily, as well as many recursive constructions.

Primitive recursive functions on $\alpha \to \beta$

As we are going for a general purpose library, we want the theory to play well with Lean. So we should be able to talk about a function $f : \alpha \to \beta$ being primitive recursive.

A bad definition, primrec functions on $\alpha \to \beta$

$f : \alpha \to \beta$ is primrec iff there exist bijections $e_1 : \mathbb{N} \to \alpha$ and $e_2 : \beta \to \mathbb{N}$ such that $e_2 \circ f \circ e_1 : \mathbb{N} \to \mathbb{N}$ is primrec.

Primitive recursive functions on $\alpha \to \beta$

As we are going for a general purpose library, we want the theory to play well with Lean. So we should be able to talk about a function $f : \alpha \to \beta$ being primitive recursive.

A bad definition, primrec functions on $\alpha \to \beta$

$f : \alpha \to \beta$ is primrec iff there exist bijections $e_1 : \mathbb{N} \to \alpha$ and $e_2 : \beta \to \mathbb{N}$ such that $e_2 \circ f \circ e_1 : \mathbb{N} \to \mathbb{N}$ is primrec.

A good definition of primrec on other types should coincide with the original when $\alpha = \beta = \mathbb{N}$, but this one implies that the halting oracle

$$f(n) = \begin{cases} 1 & \text{if the } n\text{th program halts} \\ 0 & o.w. \end{cases}$$

is primitive recursive with $e_1 = \mathrm{id}$, and $e_2(i_n) = 2n$, $e_2(j_n) = 2n + 1$ where i_n, j_n enumerate halting and non-halting programs resp.

Primitive recursive functions on $\alpha \to \beta$

- Intuitively, e_2 should not have been admissible because it's not computable, but we can't say what that means in general since $e_2 : \beta \to \mathbb{N}$.
- What we need is a "standard bijection" on all countable types so that we can coherently talk about other bijections being computable or not by comparing to the standard one
- Typeclasses to the rescue!

Primcodable types

- A type α is *encodable* if it comes equipped with maps $\text{encode} : \alpha \to \mathbb{N}$ and $\text{decode} : \mathbb{N} \to \text{option } \alpha$ such that $\text{decode } (\text{encode } a) = \text{some } a$.
  - Classically, this means the same as α is countable (finite or countably infinite).
- Encodable types are closed under most type formers
- Encodable instances can be inferred by typeclass inference
- An encodable instance yields a nontrivial function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = \begin{cases} \text{encode } a + 1 & \text{if decode } n = \text{some } a \\ 0 & \text{if decode } n = \text{none} \end{cases}$$

- A type α is *primcodable* if f is primrec.

Primitive recursive functions on $\alpha \to \beta$

A better definition, primrec functions on $\alpha \to \beta$

For primcodable types α, β, $f : \alpha \to \beta$ is primrec iff $\text{encode}_{\text{opt}\,\mathbb{N}} \circ \text{map}\ (\text{encode}_\beta \circ f) \circ \text{decode}_\alpha : \mathbb{N} \to \mathbb{N}$ is primrec.

- This definition coincides with the original for functions $\mathbb{N} \to \mathbb{N}$, using the standard primcodable instance for $\mathbb{N}$.
- The assumption that α and β are primcodable rather than just encodable is required to prove that the identity function $\alpha \to \alpha$ is primrec.
- This definition does not directly apply to binary functions $\alpha \to \beta \to \gamma$ because $\beta \to \gamma$ is not usually primcodable (it is not countable), but we can define binary primrec functions by currying

Everything interesting is primitive recursive

- There are many things in the library, but almost everything computable in the Lean sense is primitive recursive, e.g. all standard functions on option, sum, prod, list, vector, etc.
- It is not hard to get course of values recursion from this, once we know that list α is primcodable.
- Many theorems later...

Partiality

- If the goal is to get to a universal function, primitive recursion isn't enough. How to deal with partiality?
- Lean has a type of partial functions already:

$$\alpha \nrightarrow \beta := \Sigma(S : \text{set } \alpha), S \to \beta$$

Partiality

- If the goal is to get to a universal function, primitive recursion isn't enough. How to deal with partiality?
- Lean has a type of partial functions already:

$$\begin{aligned} \alpha \nrightarrow \beta &:= \Sigma(S : \mathrm{set}\ \alpha), S \to \beta \\ &\equiv \Sigma(S : \mathrm{set}\ \alpha), \{x : \alpha \mid x \in S\} \to \beta \end{aligned}$$

Partiality

- If the goal is to get to a universal function, primitive recursion isn't enough. How to deal with partiality?
- Lean has a type of partial functions already:

$$\begin{aligned} \alpha \nrightarrow \beta := & \Sigma(S : \mathrm{set}\ \alpha), S \to \beta \\ \equiv & \Sigma(S : \mathrm{set}\ \alpha), \{x : \alpha \mid x \in S\} \to \beta \\ \equiv & \Sigma(p : \alpha \to \mathrm{Prop}), \{x : \alpha \mid p\ x\} \to \beta \end{aligned}$$

Partiality

- If the goal is to get to a universal function, primitive recursion isn't enough. How to deal with partiality?
- Lean has a type of partial functions already:

$$
\begin{aligned}
\alpha \nrightarrow \beta := {} & \Sigma(S : \text{set } \alpha), S \to \beta \\
\equiv {} & \Sigma(S : \text{set } \alpha), \{x : \alpha \mid x \in S\} \to \beta \\
\equiv {} & \Sigma(p : \alpha \to \text{Prop}), \{x : \alpha \mid p\ x\} \to \beta \\
\simeq {} & \Sigma(p : \alpha \to \text{Prop}), \Pi(x : \alpha), p\ x \to \beta
\end{aligned}
$$

Partiality

- If the goal is to get to a universal function, primitive recursion isn't enough. How to deal with partiality?
- Lean has a type of partial functions already:

$$
\begin{aligned}
\alpha \nrightarrow \beta := \Sigma(S : \text{set } \alpha), S \to \beta \\
&\equiv \Sigma(S : \text{set } \alpha), \{x : \alpha \mid x \in S\} \to \beta \\
&\equiv \Sigma(p : \alpha \to \text{Prop}), \{x : \alpha \mid p\ x\} \to \beta \\
&\simeq \Sigma(p : \alpha \to \text{Prop}), \Pi(x : \alpha), p\ x \to \beta \\
&\simeq \Pi(x : \alpha), \Sigma(p : \text{Prop}), p \to \beta
\end{aligned}
$$

Partiality

- If the goal is to get to a universal function, primitive recursion isn't enough. How to deal with partiality?
- Lean has a type of partial functions already:

$$\begin{aligned}
\alpha \nrightarrow \beta &:= \Sigma(S : \text{set } \alpha), S \to \beta \\
&\equiv \Sigma(S : \text{set } \alpha), \{x : \alpha \mid x \in S\} \to \beta \\
&\equiv \Sigma(p : \alpha \to \text{Prop}), \{x : \alpha \mid p\ x\} \to \beta \\
&\simeq \Sigma(p : \alpha \to \text{Prop}), \Pi(x : \alpha), p\ x \to \beta \\
&\simeq \Pi(x : \alpha), \Sigma(p : \text{Prop}), p \to \beta \\
&\equiv \alpha \to \Sigma(p : \text{Prop}), p \to \beta
\end{aligned}$$

Partiality

- If the goal is to get to a universal function, primitive recursion isn't enough. How to deal with partiality?
- Lean has a type of partial functions already:

$$\begin{aligned}
\alpha \nrightarrow \beta &:= \Sigma(S : \mathrm{set}\ \alpha), S \to \beta \\
&\equiv \Sigma(S : \mathrm{set}\ \alpha), \{x : \alpha \mid x \in S\} \to \beta \\
&\equiv \Sigma(p : \alpha \to \mathrm{Prop}), \{x : \alpha \mid p\ x\} \to \beta \\
&\simeq \Sigma(p : \alpha \to \mathrm{Prop}), \Pi(x : \alpha), p\ x \to \beta \\
&\simeq \Pi(x : \alpha), \Sigma(p : \mathrm{Prop}), p \to \beta \\
&\equiv \alpha \to \underbrace{\Sigma(p : \mathrm{Prop}), p \to \beta}_{\mathrm{part}\ \beta}
\end{aligned}$$

Partiality

- If the goal is to get to a universal function, primitive recursion isn't enough. How to deal with partiality?
- Lean has a type of partial functions already:

$$\begin{aligned}
&\Sigma(S : \text{set } \alpha), S \to \beta \\
&\equiv \Sigma(S : \text{set } \alpha), \{x : \alpha \mid x \in S\} \to \beta \\
&\equiv \Sigma(p : \alpha \to \text{Prop}), \{x : \alpha \mid p\ x\} \to \beta \\
&\simeq \Sigma(p : \alpha \to \text{Prop}), \Pi(x : \alpha), p\ x \to \beta \\
&\simeq \Pi(x : \alpha), \Sigma(p : \text{Prop}), p \to \beta
\end{aligned}$$

$$\alpha \nrightarrow \beta := \alpha \to \underbrace{\Sigma(p : \text{Prop}), p \to \beta}_{\text{part } \beta}$$

The partiality monad

$$
\begin{aligned}
\text{part } \alpha &:= \Sigma(\mathit{dom} : \text{Prop}), \mathit{dom} \to \alpha \\
\text{return } a &:= \langle \top, \lambda\_.\ a \rangle \\
\text{assert} &: \Pi(p : \text{Prop}), (p \to \text{part } \alpha) \to \text{part } \alpha \\
\text{assert } p\, f &:= \langle (\exists h : p, (f\ h)_1),\ \lambda \langle h, h' \rangle.\ (f\ h)_2\ h' \rangle \\
\text{bind } \langle p, f \rangle\ g &:= \text{assert } p\ (g \circ f)
\end{aligned}
$$

- The type operator part is a monad in which we can assert arbitrary facts using the "assert" operation above
- Classically, this type looks the same as option α, but it does not assume decidability of the domain predicate
- This differs from a relational partiality type (e.g. $\{S : \text{set } \alpha \mid |S| \le 1\}$) in that we can evaluate the function if we can prove it is in domain

Partial recursive functions

- This turns out to be the “right” notion of partiality we need for partial recursive functions, because we can define

$$\mathrm{fix} : (\alpha \nrightarrow \alpha + \beta) \to \alpha \nrightarrow \beta$$

- This allows us to constructively define the μ-operator of partial recursive functions: $\mu n.\ p(n)$ returns the smallest $n \in \mathbb{N}$ such that $p(n)$ is true if there is one, otherwise it diverges. We can represent this as

$$\mathrm{pfind} : (\mathbb{N} \nrightarrow \mathrm{bool}) \nrightarrow \mathbb{N}$$

Partial recursive functions

```
inductive partrec : (ℕ →. ℕ) → Prop
| zero : partrec (pure 0)
| succ : partrec succ
| left : partrec (λ n, fst (unpair n))
| right : partrec (λ n, snd (unpair n))
| pair {f g} : partrec f → partrec g →
  partrec (λ n,
    f n >>= λ a, g n >>= λ b, pure (mkpair a b))
| comp {f g} : partrec f → partrec g →
  partrec (λ n, g n >>= f)
| prec {f g} : partrec f → partrec g →
  partrec (unpaired (λ a n, nat.rec_on n (f a)
    (λ y IH, IH >>= λ i,
      g (mkpair a (mkpair y i)))))
| find {f} : partrec f → partrec (λ a,
  find (λ n, (λ m, m = 0) <$> f (mkpair a n)))
```

Primitive recursive functions

```
inductive primrec : (ℕ → ℕ) → Prop
| zero : primrec (λ n, 0)
| succ : primrec succ
| left : primrec (λ n, fst (unpair n))
| right : primrec (λ n, snd (unpair n))
| pair {f g} : primrec f → primrec g →
  primrec (λ n,
    mkpair (f n) (g n))
| comp {f g} : primrec f → primrec g →
  primrec (f ∘ g)
| prec {f g} : primrec f → primrec g →
  primrec (unpaired (λ a n, nat.rec_on n (f a)
     (λ y IH,
       g (mkpair a (mkpair y IH))))
```

Partial recursive functions

```
inductive partrec : (ℕ →. ℕ) → Prop
| zero : partrec (pure 0)
| succ : partrec succ
| left : partrec (λ n, fst (unpair n))
| right : partrec (λ n, snd (unpair n))
| pair {f g} : partrec f → partrec g →
  partrec (λ n,
    f n >>= λ a, g n >>= λ b, pure (mkpair a b))
| comp {f g} : partrec f → partrec g →
  partrec (λ n, g n >>= f)
| prec {f g} : partrec f → partrec g →
  partrec (unpaired (λ a n, nat.rec_on n (f a)
    (λ y IH, IH >>= λ i,
      g (mkpair a (mkpair y i)))))
| find {f} : partrec f → partrec (λ a,
  find (λ n, (λ m, m = 0) <$> f (mkpair a n)))
```

Codes for partial recursive functions

```
inductive code : Type
| zero : code
| succ : code
| left : code
| right : code
| pair : code → code → code
| comp : code → code → code
| prec : code → code → code
| find' : code → code
```

Codes for partial recursive functions

```
inductive code : Type
| zero : code
| succ : code
| left : code
| right : code
| pair : code → code → code
| comp : code → code → code
| prec : code → code → code
| find' : code → code
```

- This is a primcodable type
- The constructors are primrec, and the recursor is also primrec (partrec) if the functions giving the cases are primrec (partrec)

Evaluation

```
def evaln : ∀ k : ℕ, code → ℕ → option ℕ := ...
def eval : code → ℕ →. ℕ := ...
```

- The function evaln k c n evaluates the function described by c at n for at most k "steps"
  - Only the `prec` and `find'` cases actually need to take a step, the others are well founded on the size of the program
  - If the function does not terminate in k steps, or if it uses values larger than k in a subcomputation, then it returns none
- eval c n evaluates c "all the way", with the partial result indicating divergence
  - eval c is the semantics corresponding to the syntax c
- evaln is primitive recursive, and eval is partial recursive. Proof: by construction
  - $\Rightarrow$ eval is a universal partial recursive function

Computability theory

The s-m-n theorem

A function $\text{curry} : \text{code} \to \mathbb{N} \to \text{code}$ satisfying $\text{eval}\ (\text{curry}\ c\ m)\ n = \text{eval}\ c\ (m, n)$ is primrec.

The fixed point theorems

1. If $f : \text{code} \to \text{code}$ is computable, then there exists some code c such that $\text{eval}\ (f\ c) = \text{eval}\ c$.
2. If $f : \text{code} \to \mathbb{N} \nrightarrow \mathbb{N}$ is partial recursive, then there exists some code c such that $\text{eval}\ c = f\ c$.

Rice's theorem

Let $C \subseteq (\mathbb{N} \nrightarrow \mathbb{N})$ such that $\{c \mid \text{eval}\ c \in C\}$ is computable. Then C is trivial, that is, $C = \emptyset$ or $C = \mathbb{N} \nrightarrow \mathbb{N}$.

Future directions

- All library objectives were achieved
  - i.e. we can say "computable" now and have the basic facts
- Turing jump (a.k.a. what problems become computable with an oracle for the halting problem?)
  - ...even less realistic than the computability theory done here
- Complexity theory
  - this requires less abstract models of computation
  - Work in this direction is much more advanced in Coq, see Forster & Smolka (2017)
- Proving equivalence of different models of computation to this one
  - Some work has been done on TM → Wang-B → partrec
  - Work in Isabelle is more advanced, see Xu, Zhang, & Urban (2013)

Future directions

- All library objectives were achieved
  - i.e. we can say “computable” now and have the basic facts
- Turing jump (a.k.a. what problems become computable with an oracle for the halting problem?)
  - ...even less realistic than the computability theory done here
- Complexity theory
  - this requires less abstract models of computation
  - Work in this direction is much more advanced in Coq, see Forster & Smolka (2017)
- Proving equivalence of different models of computation to this one
  - Some work has been done on TM → Wang-B → partrec
  - Work in Isabelle is more advanced, see Xu, Zhang, & Urban (2013)
- All of these models are too abstract; come back on Friday to see something more practical (formalizing x86)