Calculus of Inductive Constructions

Building on the Calculus of Constructions

Foundation of Software

Last week

Limitations of Pure CoC

Pure CoC has limitations:

Limitations of Pure CoC

Pure CoC has limitations:

- **Defining Data Structures:** CoC lacks direct support for defining data structures (natural numbers, lists, or trees) in a way that supports structural induction.

Limitations of Pure CoC

Pure CoC has limitations:

- **Defining Data Structures:** CoC lacks direct support for defining data structures (natural numbers, lists, or trees) in a way that supports structural induction.
- **Reasoning by Induction:** As seen in previously, proving properties like $1+n=n+1$ for Church-encoded natural numbers is very challenging (not so easy to define an induction principle).

Limitations of Pure CoC

Pure CoC has limitations:

- **Defining Data Structures:** CoC lacks direct support for defining data structures (natural numbers, lists, or trees) in a way that supports structural induction.
- **Reasoning by Induction:** As seen in previously, proving properties like 1+n=n+1 for Church-encoded natural numbers is very challenging (not so easy to define an induction principle).

The core issue: Pure CoC doesn't have a primitive mechanism for user-defined inductive types and their associated structural reasoning principles.

Introducing: Calculus of Inductive Constructions (CIC)

To address these limitations, CoC was extended to the **Calculus of Inductive Constructions (CIC)**.

- CIC enriches CoC by adding a native mechanism to define inductive types directly within the system.
- This allows for:
  - Natural definitions of data types (e.g., natural numbers, lists, booleans).
  - Direct support for structural induction and recursion.
- Proof assistants like Coq and Lean are based on variants of CIC.

Inductive Types: a simplified approach

Inductive Types

Defining an inductive type is a fundamental extension to the theory:

Inductive Types

Defining an inductive type is a fundamental extension to the theory:

- **A new type constant**: The name of the inductive type itself (e.g., `Nat`).

Inductive Types

Defining an inductive type is a fundamental extension to the theory:

- **A new type constant**: The name of the inductive type itself (e.g., `Nat`).
- **New constructor constants**: Functions that build terms of this new type (e.g., `Nat.zero`, `Nat.succ`).

Inductive Types

Defining an inductive type is a fundamental extension to the theory:

- **A new type constant**: The name of the inductive type itself (e.g., `Nat`).
- **New constructor constants**: Functions that build terms of this new type (e.g., `Nat.zero`, `Nat.succ`).
- **A new primitive elimination principle (recursor)**: A special function (e.g., `Nat.rec`) that enables deconstruction, case analysis, recursion, and induction over the type.

Inductive Types

Defining an inductive type is a fundamental extension to the theory:

- **A new type constant**: The name of the inductive type itself (e.g., `Nat`).
- **New constructor constants**: Functions that build terms of this new type (e.g., `Nat.zero`, `Nat.succ`).
- **A new primitive elimination principle (recursor)**: A special function (e.g., `Nat.rec`) that enables deconstruction, case analysis, recursion, and induction over the type.
- **New definitional computation rules (ι-reduction)**: These rules specify precisely how the recursor behaves when applied to terms built by the constructors.

💡 The core idea: Every inductive definition basically adds new primitives to the system, and new reduction rules.

Inductive Types

Defining an inductive type is a fundamental extension to the theory:

- **A new type constant**: The name of the inductive type itself (e.g., `Nat`).
- **New constructor constants**: Functions that build terms of this new type (e.g., `Nat.zero`, `Nat.succ`).
- **A new primitive elimination principle (recursor)**: A special function (e.g., `Nat.rec`) that enables deconstruction, case analysis, recursion, and induction over the type.
- **New definitional computation rules (ι-reduction)**: These rules specify precisely how the recursor behaves when applied to terms built by the constructors.

💡 The core idea: Every inductive definition basically adds new primitives to the system, and new reduction rules.

Example: Natural Numbers

Defining `Nat` in Lean

Let's define natural numbers:

```
inductive Nat where
| zero : Nat
| succ : Nat -> Nat
```

This declaration introduces the following into our ambient Calculus of Construction :

- **Type Constant:** `Nat : Type 0` .
- **Constructor Constants:**
  - `Nat.zero : Nat`
  - `Nat.succ : Nat -> Nat`
- **Primitive Recursor Constant:** `Nat.rec`.

The Recursor: `Nat.rec`

The recursor `Nat.rec` is dependently typed:

```
Nat.rec :
    forall {motive : Nat -> Sort u}
    -- What we want to define/prove for each Nat
    (zero_case : motive Nat.zero)
    -- How to handle the 'zero' case
    (succ_case : (n : Nat) ->
       (ih : motive n) ->
       motive (Nat.succ n))
    -- How to handle the 'succ n' case,
    (t : Nat)
    -- The number on which we want to compute/prove
    ,
    motive t
```

This operator enables *both* limited recursion and proof by induction.

Computation: ι-Reduction Rules for `Nat.rec`

The "new reduction rules" for `Nat` are its ι-reduction rules. These specify how `Nat.rec` computes :
Let `m` be the motive, `Z` be the `zero_case`, and `S` be the `succ_case`.

1. **Base Case Rule:**
   `Nat.rec m Z S Nat.zero` $\rightsquigarrow$ `Z`

2. **Step Case Rule:**
   `Nat.rec m Z S (Nat.succ n)` $\rightsquigarrow$ `S n (Nat.rec m Z S n)`

Defining Functions and Proving with `Nat`

User-Friendly Definitions: Pattern Matching

We are more used to pattern matching to define functions

Example: Defining addition for `Nat` :

```
def add (m n : Nat) : Nat :=
match n with
| Nat.zero   => m
| Nat.succ n’ => Nat.succ (add m n’)
```

How add could be compiled to Nat.rec

The Compiled Form:

```
Nat.rec
  (fun (k : Nat) => Nat)
  -- motive: for any Nat k, we produce a result of type nat
  m
  -- zero_case: if n is Nat.zero, result is m
  (fun (k : Nat) (ih : Nat) => Nat.succ ih)
  -- succ_case: if n is Nat.succ k,
  -- apply Nat.succ to recursive result (ih)
  n
  -- The value n we are doing recursion on
```

💡 Pattern matching can be seen as convenient syntax grounded in the primitive recursor and its ι-reduction rules.

Proving with Nat.rec: Example Induction

```
add_zero (n : Nat) : LeibnizEq (add .zero n) n :=
  @Nat.rec
   -- Motive Nat -> Prop
    (fun (x : Nat) => LeibnizEq (add .zero x) x)
    -- zero: LeibnizEq (add Nat.zero Nat.zero) Nat.zero
    (leibniz_refl (add Nat.zero Nat.zero) )
    -- succ: LeibnizEq (add Nat.zero (Nat.succ k)) (Nat.suc
    (fun (k : Nat) (ih : LeibnizEq (add Nat.zero k) k) =>
       -- We use leibniz_trans h1 h2 where:
       -- h1: add zero (succ k) = succ (add zero k)
       -- h2: succ (add zero k) = succ k
       leibniz_tran
         (leibniz_refl (add Nat.zero (Nat.succ k)))
         (leibniz_congr ih Nat.succ))
    -- The argument 'n' for which P(n) is being proved
    n
```

The problem with dependent types: Length-Indexed Vectors (`Vector` α `n`)

`Vector` α `n`: Definition

`Vector` α `n` is the type of lists of elements of type α that are statically known to have length `n`. It's an indexed inductive type.

```
inductive Vector (T : Type u) : Nat -> Type u where
| nil : Vector T .zero
| cons (head : T) {n : Nat} (tail : Vector T n)
: Vector T (.succ n)
```

We add:

- **Type Constant:** `Vector : Type u -> Nat -> Type u`.
- **Constructor Constants:**
  - `Vector.nil :` α `: Type u -> Vector` α `0`
  - `Vector.cons :` α `: Type u -> (head :` α`) -> n : Nat -> (tail : Vector` α `n) -> Vector` α `(add n 1)`
- ...

`Vector.append`: Dependent Pattern Matching

We define append using pattern matching.

```
def append {T : Type u} {n m : Nat}
(v1 : Vector T n) (v2 : Vector T m) : Vector T (add n m) :=
match v1 with
| Vector.nil => v2
| Vector.cons x xs => ...
```

Well-Typed?:

Vector.append: Dependent Pattern Matching

We define append using pattern matching.

```
def append {T : Type u} {n m : Nat}
(v1 : Vector T n) (v2 : Vector T m) : Vector T (add n m) :=
match v1 with
| Vector.nil => v2
| Vector.cons x xs => ...
```

Well-Typed?:

- **Base Case (`Vector.nil`)**:
  - If v_1 is `Vector.nil`, then its type implies `n = 0`.
  - The function must return `Vector` α `(0 + m)`.
  - We return v_2, which has type `Vector` α `m`.
  - This is not type-correct `add 0 m` and `m` do not reduce to each other!

Vector.append: Dependent Pattern Matching

We define append using pattern matching.

```
def append {T : Type u} {n m : Nat}
(v1 : Vector T n) (v2 : Vector T m) : Vector T (add n m) :=
match v1 with
| Vector.nil => v2
| Vector.cons x xs => ...
```

Well-Typed?:

- **Base Case (Vector.nil)**:
  - If v_1 is Vector.nil, then its type implies n = 0.
  - The function must return Vector α (0 + m).
  - We return v_2, which has type Vector α m.
  - This is not type-correct add 0 m and m do not reduce to each other!

```
| Vector.nil =>
  leibniz_cast_vector (leibniz_symm (add_zero m)) v2
```

Conclusion: The Inductive Power of CIC in Lean

- The philosophy for inductive types is to treat each definition as an extension of the calculus, adding:
  - New primitive constants: the type itself, its constructors, and a type-specific recursor.
  - New specific definitional computation rules: ι-reduction rules that govern how the recursor behaves with the constructors.
- This "primitive + ι-rule" approach provides a foundational mechanism for:
  - Defining data structures directly and naturally (e.g., `Nat`, `Bool`, `List`, `Vector`).
  - Performing type-safe dependent programming, where types can track properties like length (e.g., `Vector.append`).
  - Reasoning rigorously about programs and data using structural induction, directly supported by the recursor.

Conclusion: The Inductive Power of CIC in Lean

- The philosophy for inductive types is to treat each definition as an extension of the calculus, adding:
  - New primitive constants: the type itself, its constructors, and a type-specific recursor.
  - New specific definitional computation rules: ι-reduction rules that govern how the recursor behaves with the constructors.
- This "primitive + ι-rule" approach provides a foundational mechanism for:
  - Defining data structures directly and naturally (e.g., `Nat`, `Bool`, `List`, `Vector`).
  - Performing type-safe dependent programming, where types can track properties like length (e.g., `Vector.append`).
  - Reasoning rigorously about programs and data using structural induction, directly supported by the recursor.

Are every inductive types ok? No, only "positive" recursive types. For more details, we recommend the paragraph "General Rules" page 6, of Christine Paulin-Mohring, *Introduction to the Calculus of Inductive Constructions*