

Lean言語は 新世代の純粹関数型言語になれるか？

...

Proxima Technology 井上亜星

Lean 言語ってなに？

2013年ごろに当時 Microsoft Research に在籍していた Leonardo de Moura 氏により開発が始められた定理証明支援系。

Rocq(旧Coq) や Agda と同様に、依存型という強力な型システムのパワーによって命題を表現し、証明することを可能にしている。

現在、Lean 4 がアクティブに開発中。



他の定理証明支援系とどう違うの？

大きく分けて2つの特徴があります

1. 依存型に基づく定理証明支援系であると同時に、
 純粋関数型言語でもある
 (プログラミングと証明を両立するための工夫もある)
2. 強力なメタプログラミングフレームワークを備えている

純粹？Haskell みたいな言語ってこと？

確かに Haskell にかなり影響を受けている
(モナドや do 構文もある)

しかし、Haskell にはなかったおもしろい機能がある：

- notebook なしでもフィードバックが即座に得られる
- **do** 構文が強力で、for ループや while ループも書ける
- **フィールド記法**があって、関数適用をフィールドアクセスのように書ける
- 効率的に計算ができるようにする **Functional but in-place** という仕様がある

フィードバックが即座に得られる

エディタ上で編集することにより実行結果を確認できる

画像では Error Lens 拡張機能を使用しています

```
115 ✓ def frac (n : Nat) : Nat :=  
116   · match n with  
117   · | 0 => 1  
118   · | n + 1 => (n + 1) * frac n  
119  
120 #eval frac 5    120
```

強力な do 構文

- for ループと while ループ
- let mut で可変なローカル変数
- continue と break

詳細は 'do' unchained という論文を参照

```
/-- `n`以下の素数のリストを `Array Bool` の形で返す。  
`i` 番目が `true` ならば `i` は素数で、`false` ならば合成数。 -/  
def eratosthenesAux (n : Nat) : Array Bool := Id.run do  
  let mut isPrime := Array.replicate (n + 1) true  
  
  isPrime := isPrime.set! 0 false  
  isPrime := isPrime.set! 1 false  
  
  for p in [2 : n + 1] do  
    if not isPrime[p]! then  
      continue  
  
    if p ^ 2 > n then  
      break  
  
    -- `p` の倍数を消していく  
    let mut q := p * p  
    while q ≤ n do  
      isPrime := isPrime.set! q false  
      q := q + p  
  
  return isPrime
```

フィールド記法

T が e の型であるときに、関数適用 $T.f\ e$ を $e.f$ と書くことができる

```
/-- 平面 -/  
structure Point ( $\alpha$  : Type) where  
  x :  $\alpha$   
  y :  $\alpha$   
  
-- `Point` のフィールドへのアクセサ関数  
#check Point.x  
#check Point.y  
  
-- フィールド記法を使ってアクセスすることができる  
#guard  
  let p : Point Nat := { x := 1, y := 2 }  
  p.x = Point.x p
```

Functional but in-place

他で参照されていない値を更新するとき、自動で破壊的な更新が行われる
(参照カウントに基づく)

```
/-- フィボナッチ数列を計算する -/  
def fibonacci (n : Nat) : Array Nat := Id.run do  
  -- 可変な配列 `fib` を宣言する  
  let mut fib : Array Nat := Array.mkEmpty n  
  fib := fib.push 0  
  fib := fib.push 1  
  for i in [2:n] do  
    -- 配列 `fib` のメモリアドレスを出力する  
    dbg_trace unsafe ptrAddrUnsafe fib  
  
    -- `fib` を更新  
    fib := fib.push (fib[i-1]! + fib[i-2]!)  
  
  return fib  
  
-- コピーが行われていれば別のメモリアドレスが出力されるはずだが...?  
#eval fibonacci 10
```


他の定理証明支援系とどう違うの？

大きく分けて2つの特徴があります

1. 定理証明支援系であると同時に、純粋関数型言語でもある
(プログラミングと証明を両立するための工夫もある)
2. 強力なメタプログラミングフレームワークを備えている

プログラミングと証明を両立するために

実装と論理モデルを分離することができる

- ・自然数 **Nat** は、論理モデルはペアノの公理で与えられているが、
実際の計算は別の高速な方法で行われる
- ・文字列 **String** は **Char** のリストとして定義されているが、
実際の計算は別の高速な方法で行われる

などなど...

正当性の証明付きで置換することもできるし、証明なしで置換することもできる

他の定理証明支援系とどう違うの？

大きく分けて2つの特徴があります

1. 定理証明支援系であると同時に、純粋関数型言語でもある
(プログラミングと証明を両立するための工夫もある)
2. 強力なメタプログラミングフレームワークを備えている

強力なメタプログラミングフレームワークとは？

syntax コマンドと declare_syntax_cat コマンドでパーサを書き換えて新しい構文を定義することができるほか、macro_rules コマンドでマクロも自由に書ける


```
/-- リスト内包表記 -/  
declare_syntax_cat compClause  
  
syntax "for " term " in " term : compClause  
syntax "if " term : compClause  
syntax "[" term " | " compClause,* "]" : term  
  
macro_rules  
| `([ $t | ]) => `([ $t ])  
| `([ $t | for $x in $xs ]) => `(List.map (fun $x => $t) $xs)  
| `([ $t | if $x ]) => `(if $x then [ $t ] else [])  
| `([ $t | $c, $cs,* ]) => `(List.flatten [[ $t | $cs,* ] | $c])  
  
-- for 構文のテスト  
#guard [x ^ 2 | for x in [1, 2, 3, 4, 5]] = [1, 4, 9, 16, 25]  
  
-- 2重の for 構文のテスト  
#guard  
let lhs := [(x, y) | for x in [1, 2, 3], for y in [4, 5]]  
let rhs := [(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]  
lhs = rhs
```

実際にプログラムを書けるのか？

実際に Lean で書かれた(広く使われている)プログラムの例として、

ドキュメント生成ツール **doc-gen4** や文書作成ツール **verso** がある。

実際、Lean の公式マニュアルは verso を使って書かれている。

The Lean Language Reference

▼ Table of Contents

1. Introduction

2. Elaboration and Compilation

3. Interacting with Lean

4. The Type System

5. Source Files and Modules

6. Namespaces and Sections

7. Definitions

8. Axioms

9. Attributes

10. Terms

11. Type Classes

12. Coercions

13. Tactic Proofs

14. Functors, Monads and do-Notation

15. IO

16. The Simplifier

17. Basic Propositions

18. Basic Types

19. Notations and Macros

20. Run-Time Code

21. Build Tools and Distribution

Release Notes

Index

1. Introduction →

The Lean Language Reference

This is the *Lean Language Reference*. It is intended to be a comprehensive, precise description of Lean: a reference work in which Lean users can look up detailed information, rather than a tutorial intended for new users. For other documentation, please refer to the [Lean documentation overview](#). This manual covers Lean version 4.20.0-rc2.

Lean is an **interactive theorem prover** based on dependent type theory, designed for use both in cutting-edge mathematics and in software verification. Lean's core type theory is expressive enough to capture very complicated mathematical objects, but simple enough to admit independent implementations, reducing the risk of bugs that affect soundness. The core type theory is implemented in a minimal [kernel](#) that does nothing other than check proof terms. This core theory and kernel are supported by advanced automation, realized in an [expressive tactic language](#). Each tactic produces a term in the core type theory that is checked by the kernel, so bugs in tactics do not threaten the soundness of Lean as a whole. Along with many other parts of Lean, the tactic language is user-extensible, so it can be built up to meet the needs of a given formalization project. Tactics are written in Lean itself, and can be used immediately upon definition; rebuilding the prover or loading external modules is not required.

Lean is also a **pure functional programming language**, with features such as a run-time system based on reference counting that can efficiently work with packed array structures, multi-threading, and monadic `IO`. As befits a programming language, Lean is primarily implemented in itself, including the language server, build tool, *elaborator*, and tactic system. This very book is written in [Verso](#), a documentation authoring tool written in Lean.

Familiarity with Lean's programming features is valuable even for users whose primary interest is in writing proofs, because Lean programs are used to implement new tactics and proof automation. Thus, this reference manual does not draw a barrier between the two aspects, but rather describes them together so they can shed light on one another.

まとめ

Lean 言語というプログラミング言語があって、

- Rocq (旧 Coq) のように定理証明支援系として使える
- Haskell のように純粋関数型言語としても使える

後発言語らしい工夫もあり

- 定理証明とプログラミングを両立する工夫
- 純粋関数型言語のパフォーマンス面・UI面双方の欠点を改善

自己紹介

名前: 井上亜星

所属: Proxima Technology  Proxima

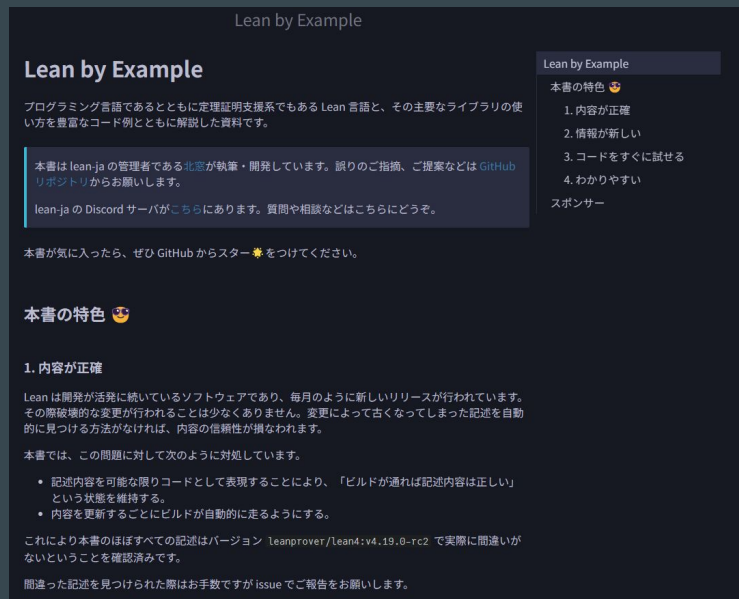
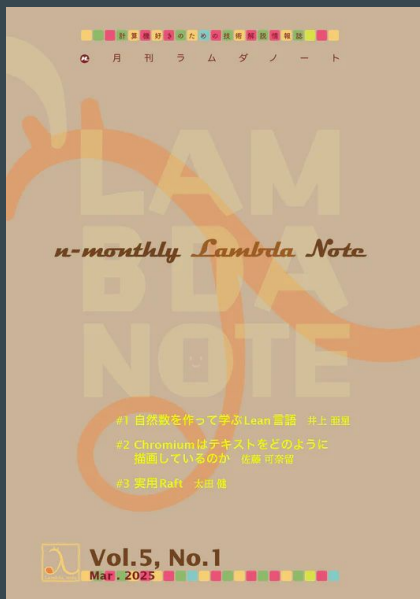
GitHub 等ネットでは 北窓 (きたまど)
という名前でやってます



自己紹介

n 月刊ラムダノートに記事を寄稿したほか、

Lean by Example というオンラインリファレンスを書いてます



おわりに

lean-ja という Lean の日本語コミュニティ

(Discordサーバ)もあります。

ご興味があればぜひ。

ご清聴ありがとうございました