IT技術とCoqの世界

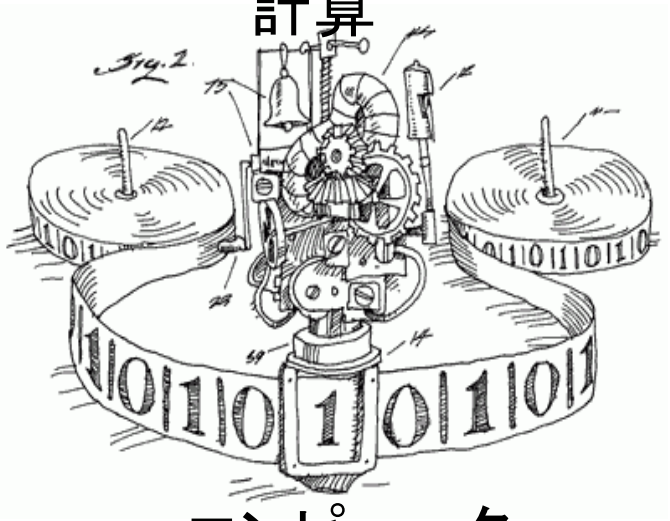
「証明」＝「プログラム」＝「計算」の意味を考える

数学者

プログラム

プログラマ

計算



コンピュータ

Most working engineers view machine-checked mathematical proofs as an academic curiosity, if they have ever heard of the concept at all. In contrast, activities like testing, debugging, and code review are accepted as essential. They are woven into the lives of nearly all developers

-- Adam Chlipala

Coming Soon: Machine-Checked Mathematical Proofs in Everyday Software and Hardware Development

-- *Adam Chlipala*

はじめに

- 筆者は、これからは、開発者にとってCoqを学ぶことが重要になるだろうと考えている。小論はその理由と背景を、まとめたものである。
- 冒頭の「なぜ、開発は人間によって担われているのか？」は、人間中心の複雑な構造を持つ開発の世界は、コンピュータの論理的推論能力の利用で、大きく変わるだろうという展望を述べている。
- 現在では、多くの人にとって、それは非現実的なビジョンだと思われるかもしれない。ただ、それは、バグのないセキュアなソフトを作るという社会的に重要な課題に対して、もっとも根本的な解答を与えるものである。同時に、それは、開発コストの大幅な削減という経済合理性を備えている。Coqは、そうした見通しを実現する上で欠かせないツールになろうとしている。

はじめに

- 次章は、先に見た開発過程の刷新の鍵であるコンピュータの「論理的＝数学的推論能力」をテーマにしている。「機械が論理的に推論する」とか「機械が数学的証明を行う」という言葉が腑に落ちない人は多いと思う。
- 「計算」「証明」「プログラム」という三つの言葉は、もともと別の意味を持っている。それは今でも同じだ。この章は機械の能力を人間がどう認識してきたかを、この三つの概念の関係の認識の歴史をメイン・ストーリーにして述べたものである。
- まず、「計算」と「証明」が、本質的には同じものであるという認識が生まれ、ついで、「プログラム」は「証明」とみなせるという認識が生まれる。理論的には、この二つの発見で、三項の関係は明確になった。「コンピュータは、プログラムの実行という形で、証明を計算する」のである。

はじめに

- さいわいなことに、現代人は、「プログラム」と「計算」の関係については、十分な直感を持っている。この直感に依拠すれば、「チャーチ＝チューリングのテーゼ」や「カリー＝ハワード対応」に遡及しなくても、三項の関係が示す、「機械が論理的・数学的推論能力を持つ」という直感が得られるのではというのが、僕が期待しているところである。
- 残念ながら、「プログラム」と「計算」の関係だけでは、「証明」へのリンクが欠けている。僕のもう一つの期待は、Coqでの証明構成の経験が、このリンクを作ってくれるだろうということである。
- これまで見てきたような、人間には何ができ、機械には何ができるのかという問題は、人工知能論の大きなテーマである。ただ、今回は、こうした問題について触れることができなかった。いずれ、機会を改めて、筆者の考えを述べたいと思う。

はじめに

- 第二部「Coqの世界」 は、二つの章からなる。

- 最初の「Coqとのはじめての対話」は、Coqに触れたことのない人に、擬似的にCoqの対話的証明構築環境を体験してもらうことを目的にしている。
- ただ、望むらくは、疑似的にではなく実際にCoqに触れてみてほしいと思っている。丸山は、自習用の Coq チュートリアルを公開している。
  https://github.com/maruyama097/coq-tutorial
- また、12月14日には、二度目のCoq ハンズオンを開催予定である。

はじめに

- 次章の「Coqによる形式的仕様の記述例」は、実際に、Coqではどのように仕様を記述するのかの紹介である。Coqによる仕様の「形式的記述」に、今一つイメージが掴めない人もいると思う。「形式的」というが、実際にCoqプログラムとして、コンピュータ上で動く仕様である。
- 紹介するのは、ひとつは、簡単な電卓ライクなスタックマシンの仕様記述例である。もう一つは、Deep Specification の代表的な取り組みの一つである。ChlipalaのKamiプロジェクトからのサンプルである。

Agenda Part I　開発の世界を考える

- なぜ、開発は人間によって担われているのか？
- 機械は論理的＝数学的推論能力を持つ
  - 「証明」＝「計算」
  - 「証明」＝「プログラム」
  - 「プログラム」＝「計算」
- 人間と機械の関係を考える

Agenda Part II　Coqの世界

- Coqとのはじめての対話
  - 人間はCoqに何を伝えたか？
  - Coqは人間に何を伝えたか？
- Coqによる形式的仕様の記述例
  - 単純なStack Machineの例
  - Deep Specificationでの仕様定義
  - Kami：仕様定義サンプル

第一部

開発の世界を考える

なぜ、開発は人間によって担われているのか？

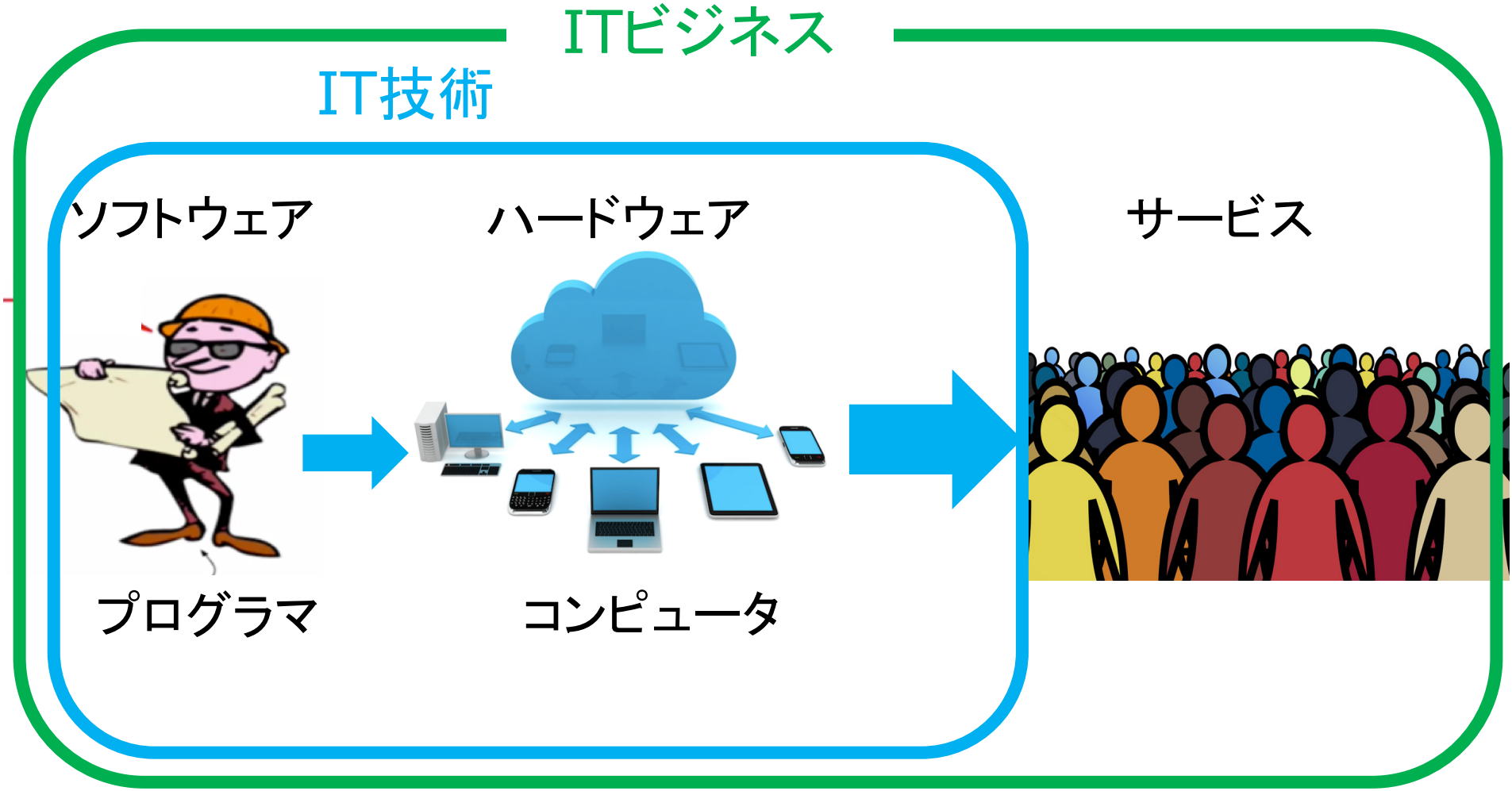
人間がしていることと、コンピュータがしていること

ITビジネスとは、単純化していうと、ソフトウェアを
ハードウェア上で動かして、それをサービスとして
顧客に提供するビジネスである

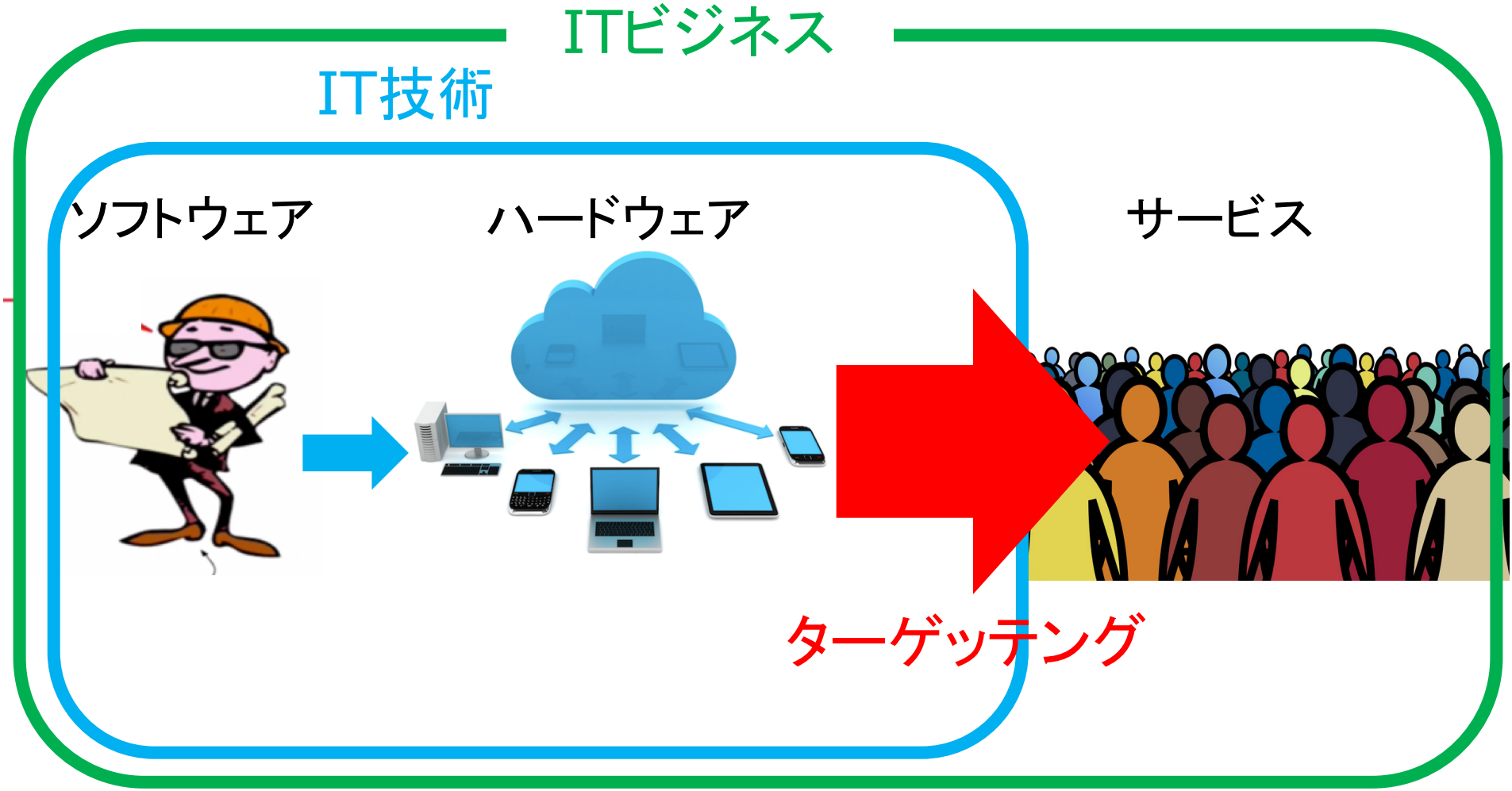
ITビジネス

ソフトウェア　　ハードウェア　　サービス

ITビジネスの中核部分は、ソフトとハードからなる
IT技術である。



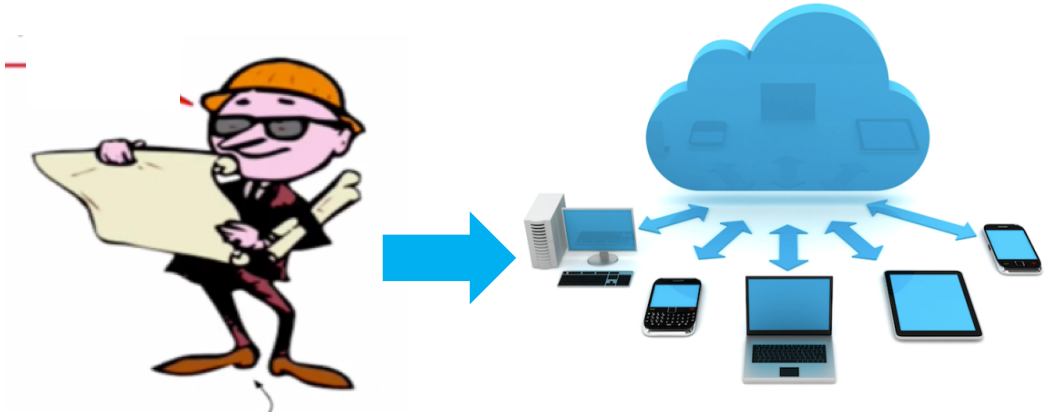
現代のITビジネスの最も大きな関心の一つは、
消費者へのサービスのターゲッテングに、IT技術（例えば、「AI技術」）を利用することなのだが、それについては別に述べる。



ここでは、ソフトウェアとハードウェアからなるIT技術の
単純なモデルから出発して、人間がしていることと
コンピュータがしていることを考えよう

IT技術

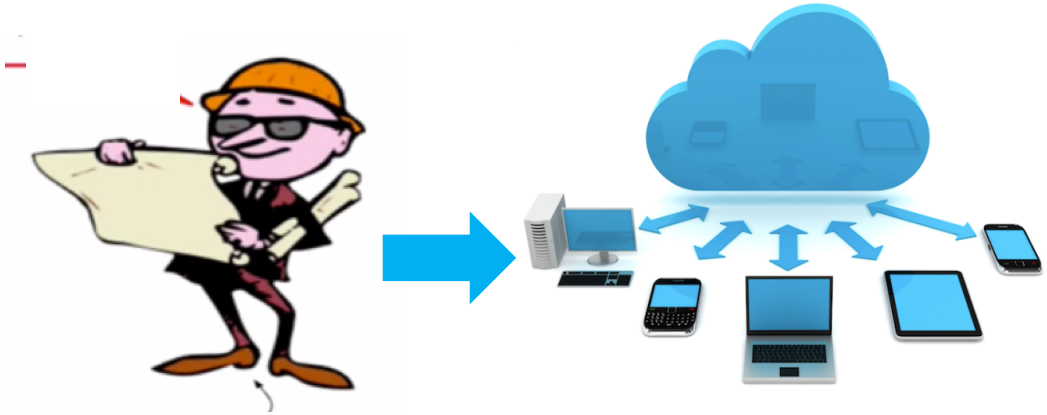
ソフトウェア　　ハードウェア



単純に考えれば、プログラマがプロプラムを作成し、
コンピュータがそれを実行する。人間がソフトを作り
コンピュータがそれを実行する。それは間違っていない。

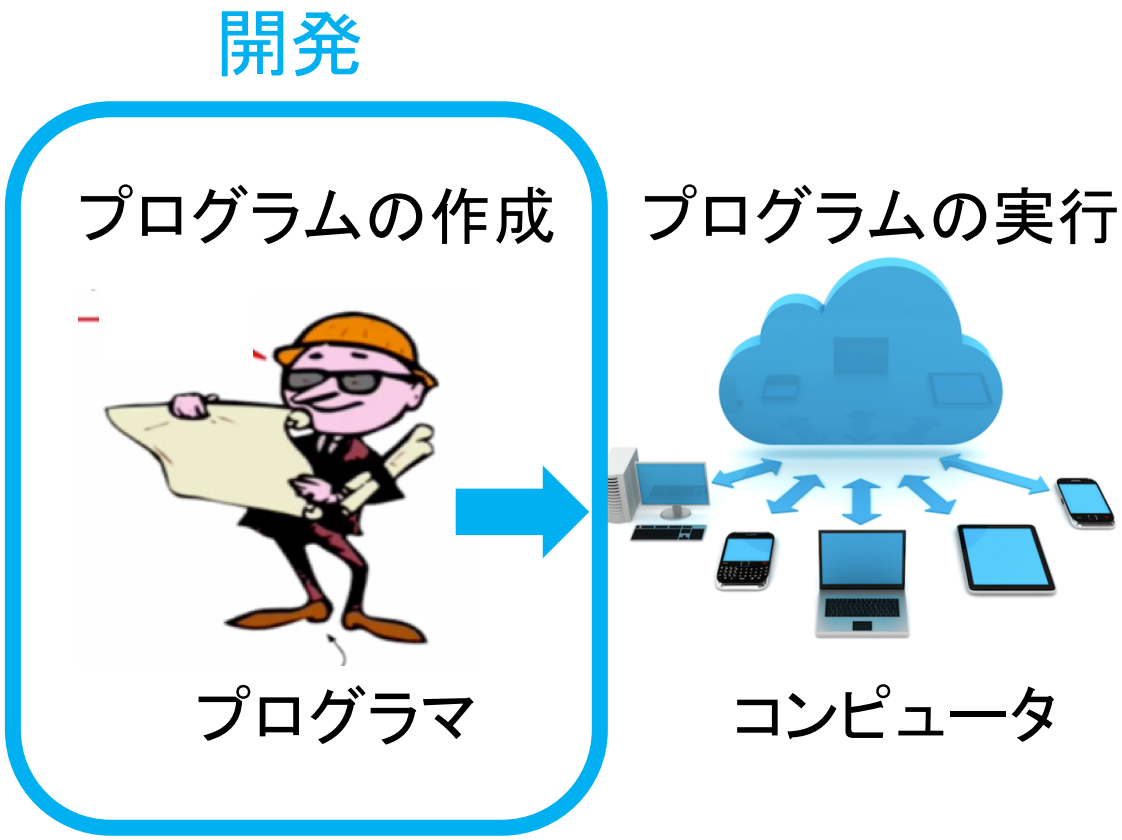
IT技術の単純なモデル

プログラムの作成　　プログラムの実行



プログラマ　　コンピュータ

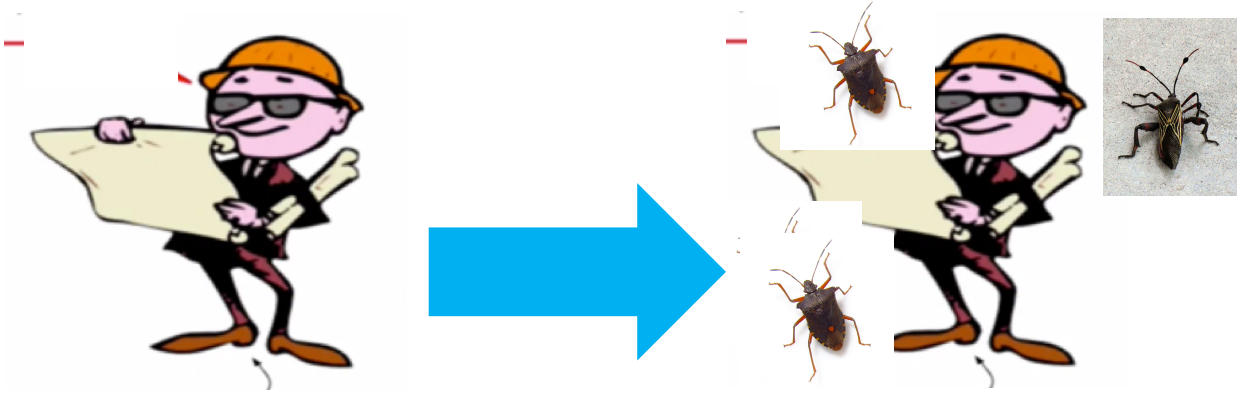
コンピュータとプログラムの関係は、シンプルである。規模の違いはあっても、それは、均一にスケールする。
それに対して、プログラマによるプログラム作成の過程（開発過程）は、複雑である。以下、それを見ていこう。



バグのないプログラムが、すぐにできるわけではない。
開発では、プログラムのテストは欠かせない。

開発

プログラムの作成　　プログラムのテスト



プログラマ　　プログラマ

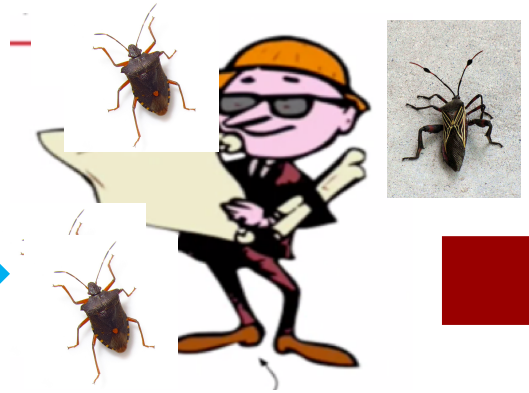
テストでバグが見つかれば、デバッグをする

開発

プログラムの作成

プログラマ

プログラムのテスト



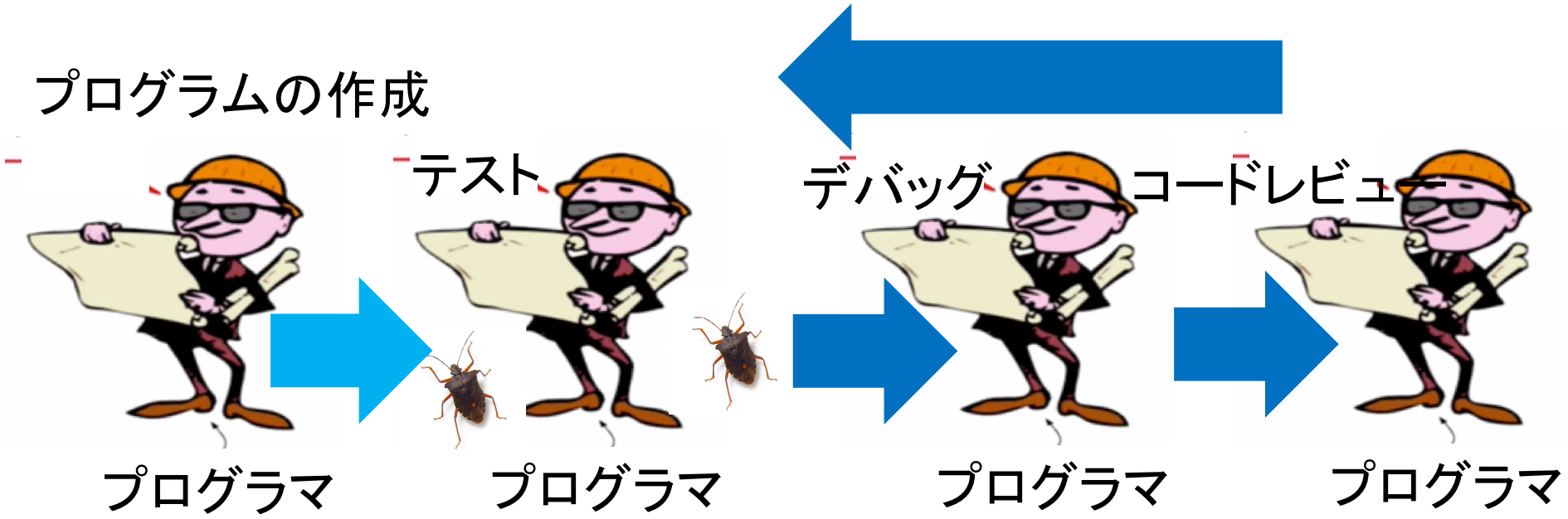
プログラマ

プログラムのデバッグ

プログラマ

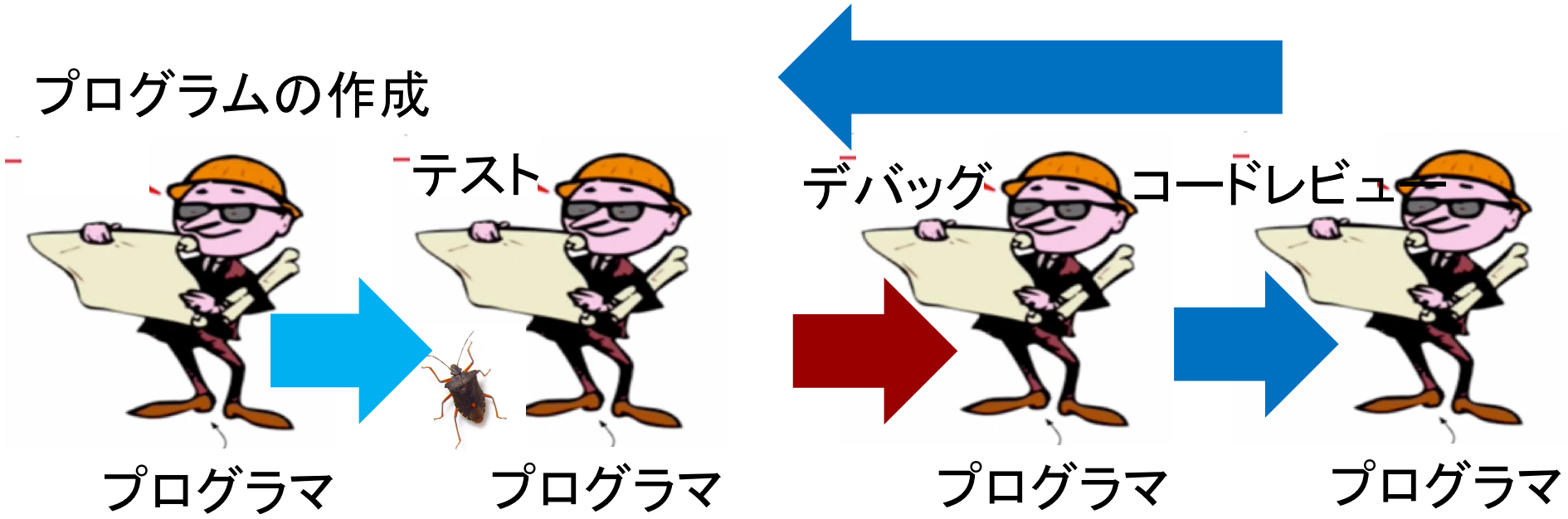
コードを見直したら、またテストをする

開発



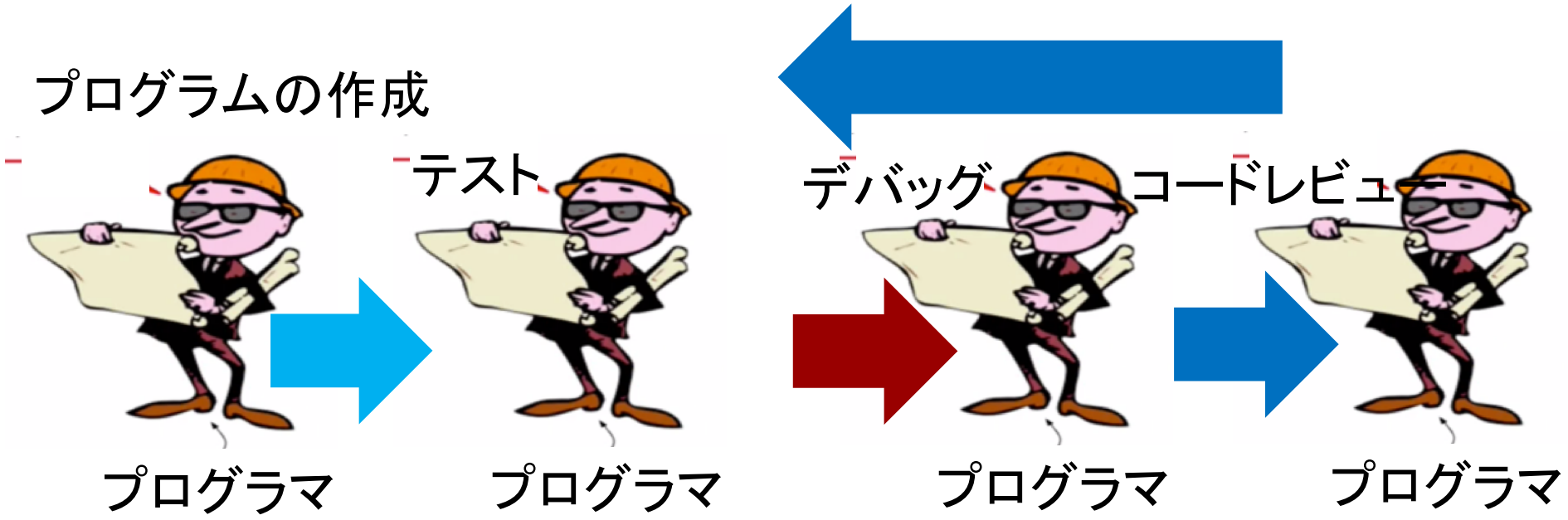
その作業は、バグがなくなるまで繰り返される

開発



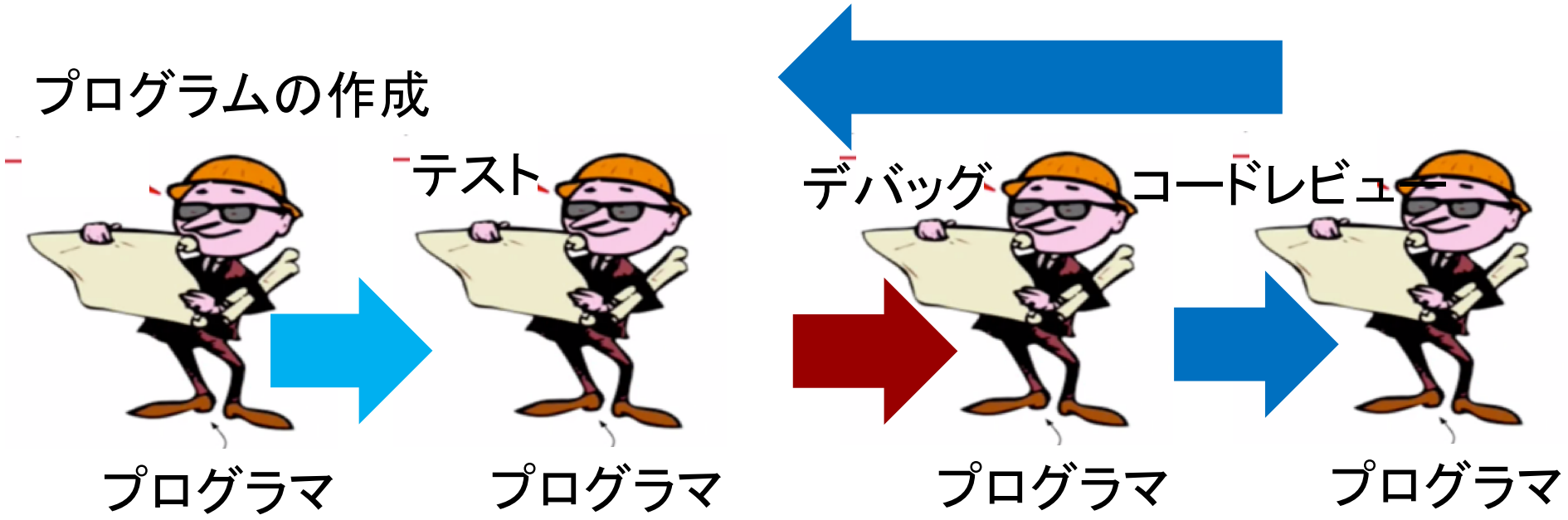
その作業は、バグがなくなるまで繰り返される

開発

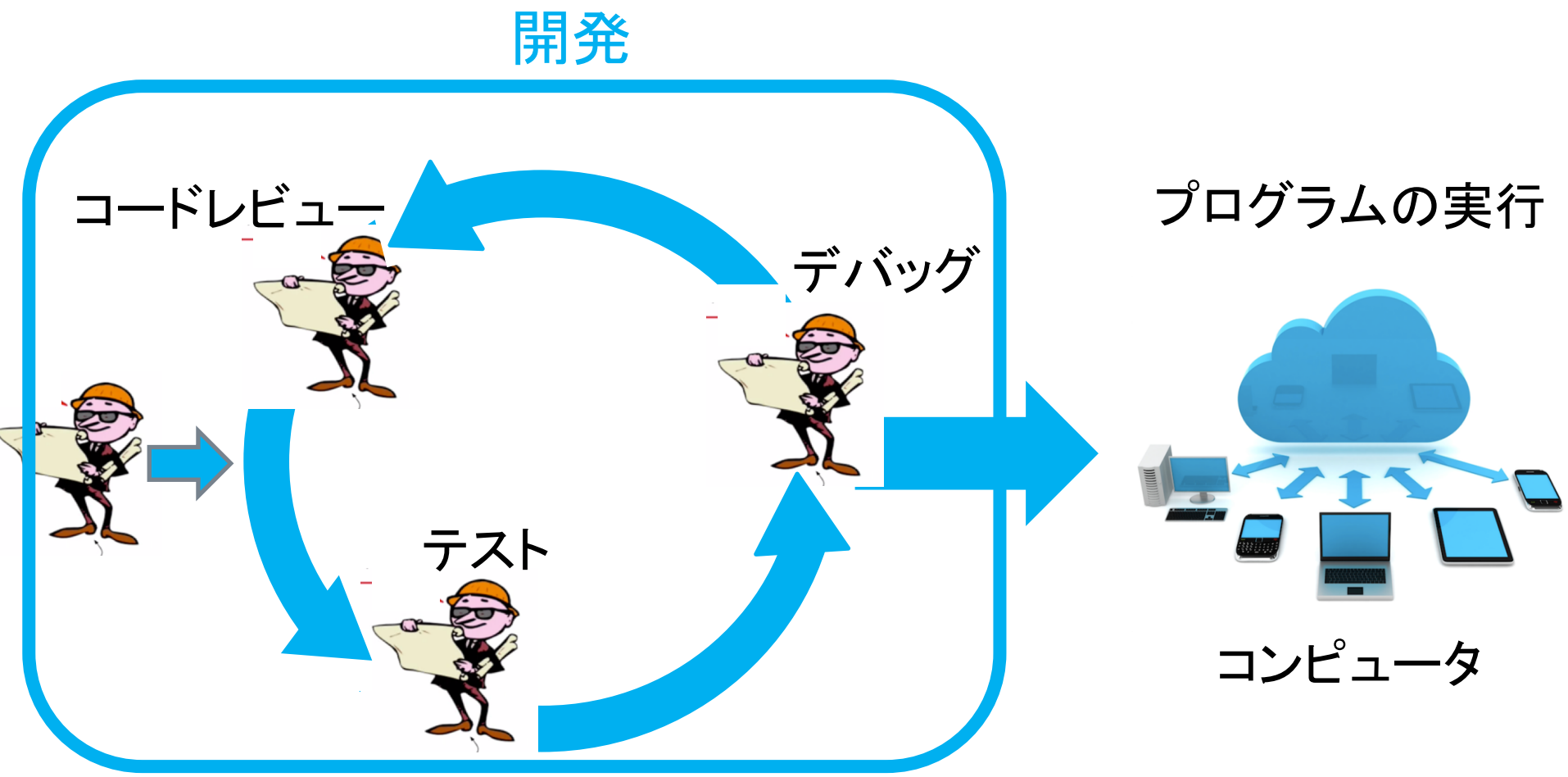


その作業は、バグがなくなるまで繰り返される

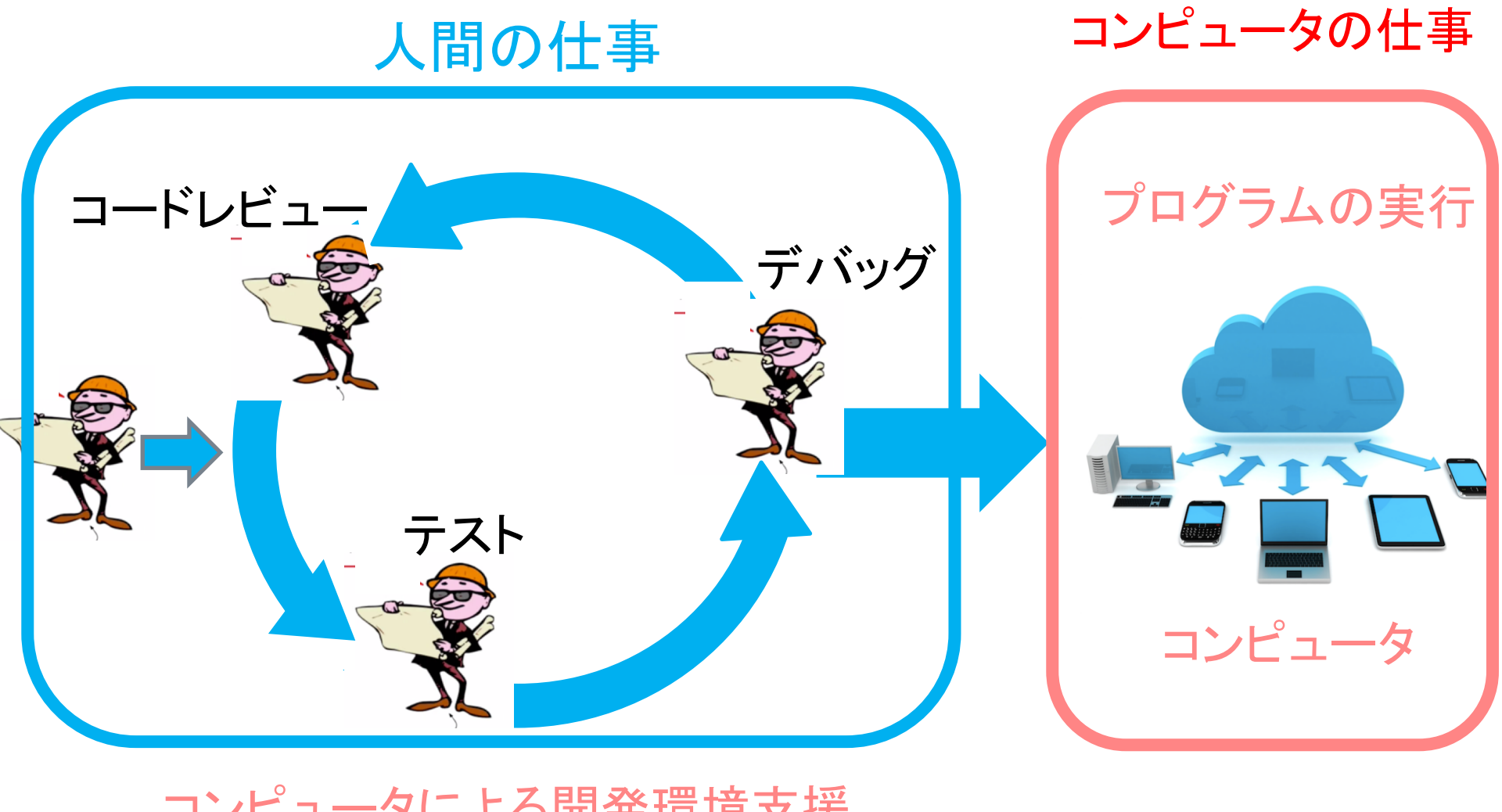
開発



開発では、こうしたテスト・デバッグ・コードレビューのサイクルが繰り返され、その後、コンピュータ上にデプロイされる。



統合エディター等、コンピュータによる開発支援環境は、現代の開発には欠かせないのだが、開発の過程は、基本的には、すべて人間の能力によって担われている。



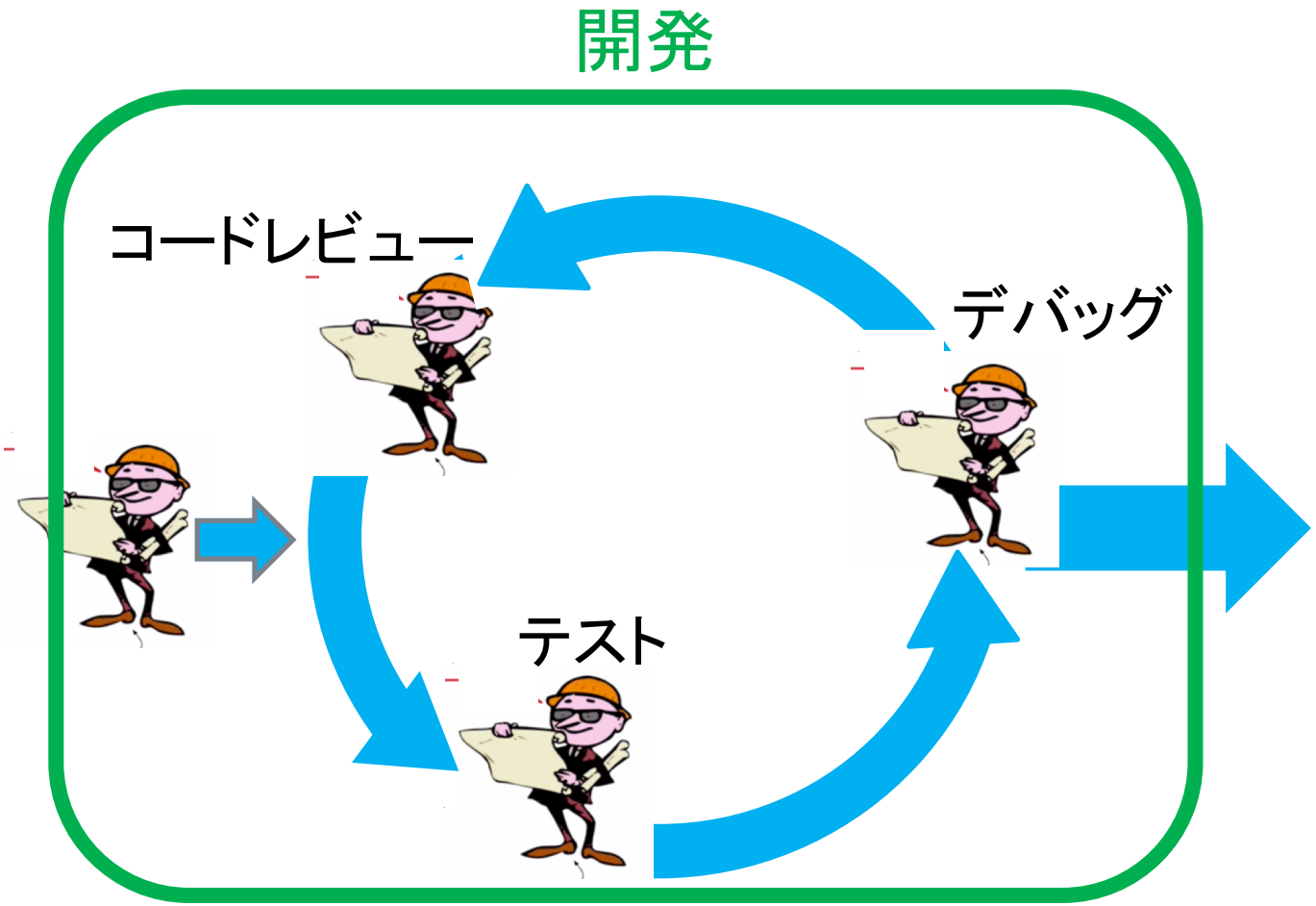
なぜ、開発は人間によって担われているのか？

- それは、ある目的を持ったプログラムを作成する能力が人間にしかないからである。人間は、機械にはない「創造的」な能力を持っている。
- ではなぜ、人間はバグを生み出すのか？
  それは、人間は「創造的」かもしれないが、「完全」ではないからである。
- ただし、人間はバグを修正できる。
  機械は、現在の技術では、バグの修正はできない。

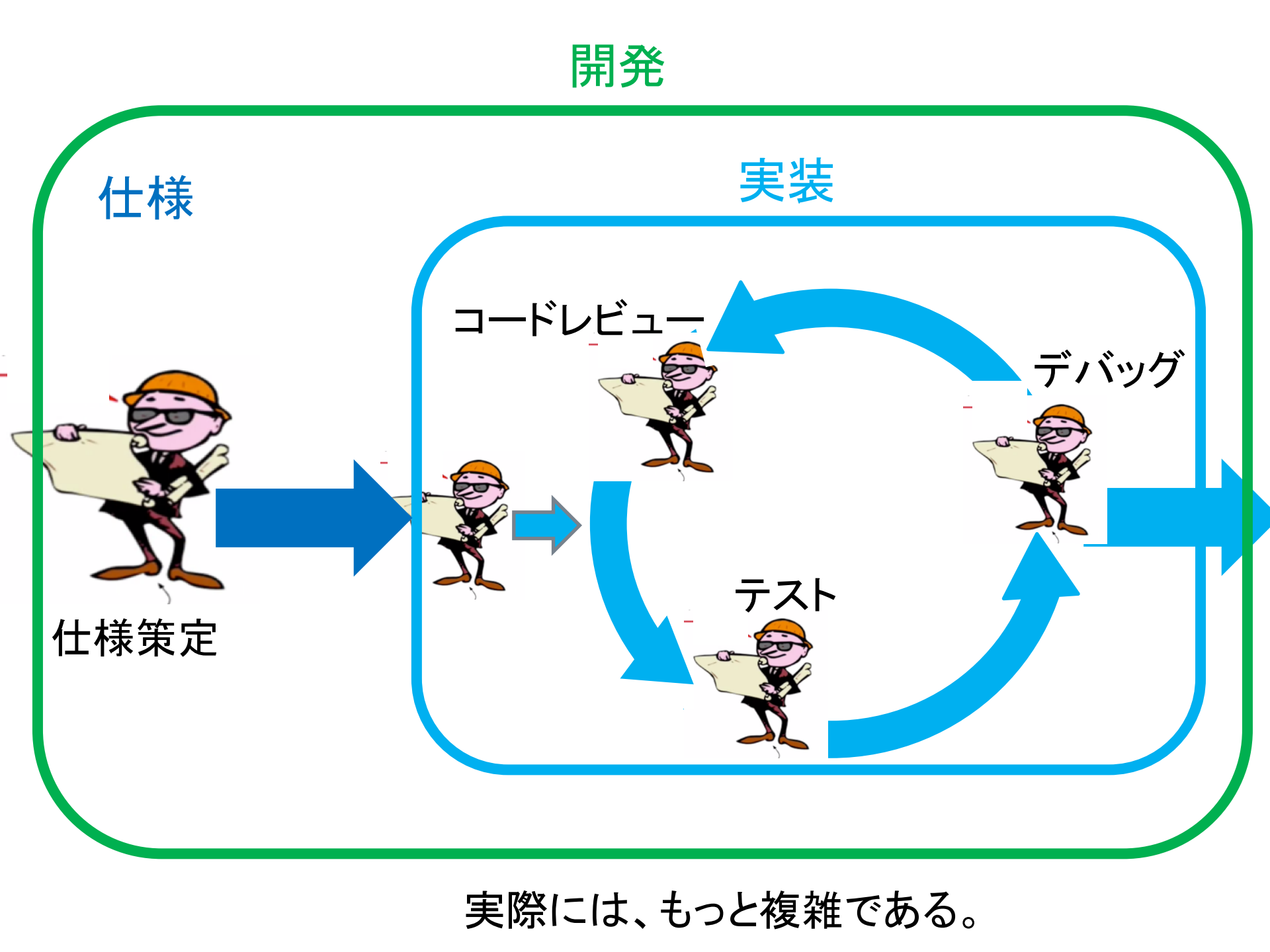
バグを修正する人間の能力はどのようなものか？

- それでは、バグを修正するのに、人間のどんな能力が使われているのだろうか？
- それはプログラムの目的に照らして、その振る舞いが正しいか否かyesかnoで判断し、正しくないなら、その原因を見つけだすという能力である。前者と後者では、多少、難しさは違う。ただ、両者とも、「論理的」に推論する能力と言っていいい。
- そこで問題となっているのは、プログラムの意図を記述する「仕様」と「実装」の関係である。
- 「仕様」の「意味」を理解することが、機械には難しいことが、ここで人間が必要とされる大きな背景になる。

先に見た、次のような開発のモデルには、「仕様」のモメントが抜けている。個人が行う単純な開発にも、それがドキュメントにはなっていなくても、「仕様」にあたるものは必ず存在する。



次のようになる



実際には、もっと複雑である。

開発

仕様

仕様策定

実装

コードレビュー

デバッグ

テスト

コードレビュー

デバック

テスト

コードレビュー

デバッグ

テスト

コードレビュー

デバッグ

テスト

なぜ、開発は人間によって担われているのか？

- 開発は、単に、人間によって担われているというだけではなく、「非常にたくさんの人間」によって担われている。
- それは、そうした作業を、現状の技術では、機械によっては代替できないからである。
- ただ、それは本当だろうか？

コンピュータが論理的推論能力を持てば、開発はどう変わるか？

- 今回のセミナーは、コンピュータが論理的推論能力を持てば、開発がどのように変わるかを考えることをテーマにしている。

- 現在の開発作業の中核をなす「テスト・デバッグ・コードの見直し」というループの繰り返しをドライブしているのは、仕様との関係でのプログラムの見直しである。
- デバッグ作業でプログラマは、プログラムとにらめっこしているように見えるのだが、その時、無意識的にでも（「仕様」はドキュメントに書かれたものだけではない。「暗黙の常識」も「仕様」として機能する）、意識的にでも、常に意識しているのは、仕様とプログラムの関係である。

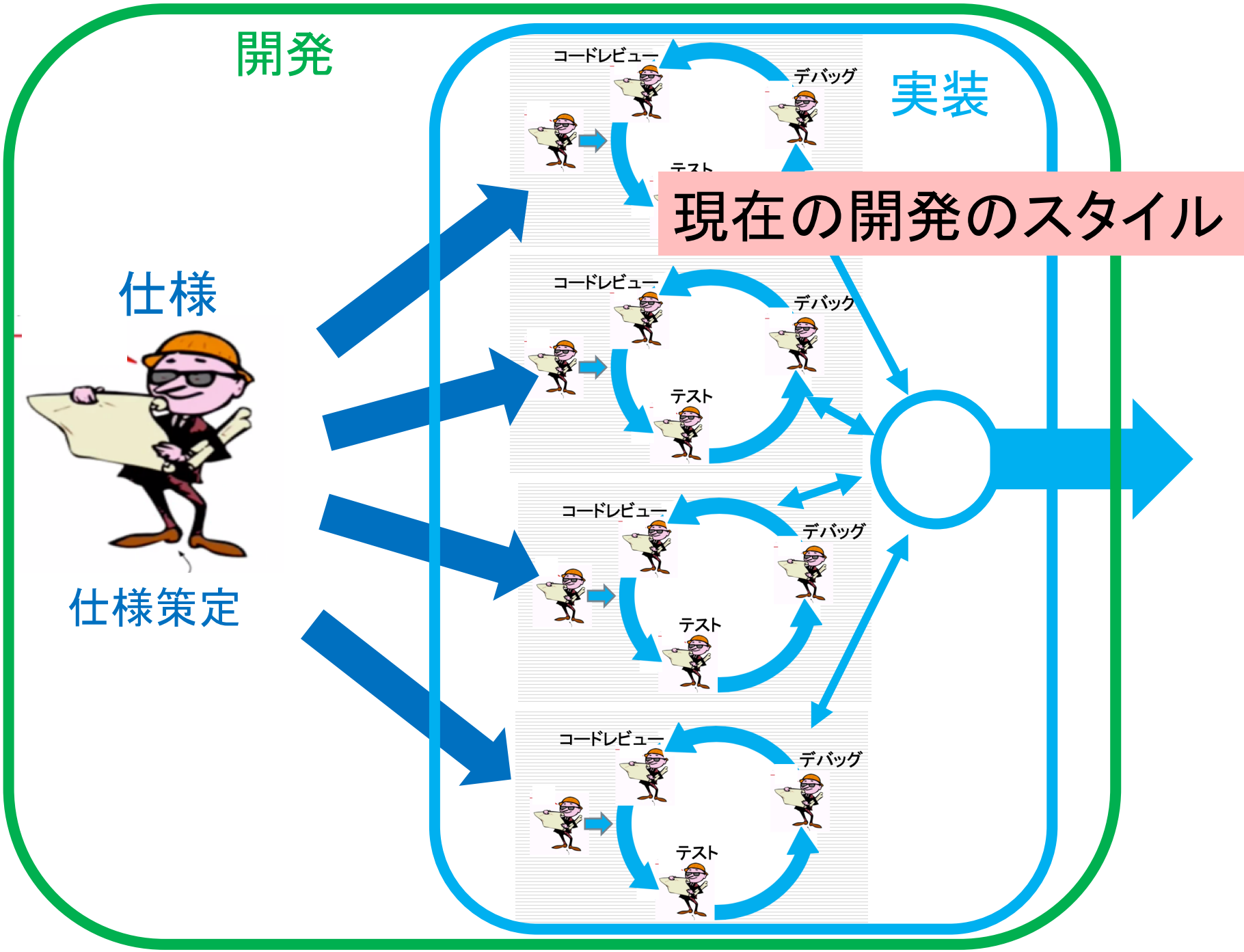
「仕様」と「実装」の関係は、論理的である

- 重要なことは、「仕様」と「実装」の関係は、論理的なものであるということである。
- もし、仕様が形式的に定義されているなら、ある実装がその形式的仕様を満たしているか否かは、形式的に証明可能である。
- もし、機械が論理的推論能力を持てば、そうした証明の実行によって、仕様と実装の一致をチェックするテスト・デバッグ作業に、人間は不要になる。

「実装」の正しさの鍵は、「仕様」が握っている

- 明らかに、「実装」が正しいか否かの判断の鍵は、「仕様」が握っている。
- ただ、「実装」が、どの実装でもコンピュータで受理されるプログラム言語の形式をとっているのに対して、「仕様」の記述のレベルは、まちまちである。仕様の策定者は、一般には、実装コードを書かない。
- もし、コンピュータが論理的推論を実行する能力を持ち、仕様と実装の一致を形式的に証明できるなら、ほとんど同じ技術を使って、形式的仕様から実装のプログラムが自動生成できる。

- コンピュータによる、仕様からのプログラムの自動生成は、開発の世界を大きく変える。


開発
実装
現在の開発のスタイル
仕様
仕様策定
コードレビュー
デバッグ
テスト
コードレビュー
デバック
テスト
コードレビュー
デバッグ
テスト
コードレビュー
デバッグ
テスト

開発

仕様

実装

将来可能な開発のスタイル

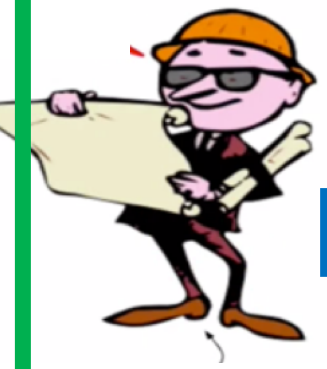
仕様策定

コンピュータによる
仕様からのプログラムの
自動生成

人間の仕事

コンピュータの仕事の拡大

仕様



仕様策定

実装

コンピュータによる
仕様からの
プログラムの
自動生成

プログラムの実行

コンピュータ

コンピュータによる仕様開発環境

開発での人間の仕事はなくなるのか？

- なくならない。

- プログラムを作成するのに、人間の「創造性」が必要だったように、形式的な仕様を作成するのには、人間の「創造的」な力がいる。
- ただ、人間による誤りの混入を防ぐために、仕様作成の各ステップでの「論理性チェック」のサポートが必要になる。それは、現在のGUIを備えた統合開発環境と同じようなものだ。コンピュータが、仕様作成を支援する。

こうした見通しは実現可能か？

- 現在では、まだ、こうした環境は部分的にしか実現されていない。こうした見通しが、開発全体に広がる可能性はあるのであろうか？
- 十分にある。

1. セキュアで堅牢なソフトウェアを提供する上で、人間から構成された開発体制の構造的複雑さは、バグの温床になる。抜本的対応がいずれ求められることになる。
2. 開発体制の複雑さは、開発期間と開発コストの増大に結びついている。開発の多くの部分の自動化には、経済合理性がある。
3. こうした開発を可能にする「証明支援システム」が登場し、その利用も大きく拡大している。

機械は論理的＝数学的推論能力を持つ

理論的背景

「証明」＝「プログラム」＝「計算」

機械の論理的＝数学的推論能力

ここでは、機械が論理的＝数学的推論能力を持つという認識がどのように形成されてきたのかを、歴史的・理論的に振り返りたいと思う。

こうした振り返りは、なぜこうした認識が今日まで十分に確立されなかったのか、その理由も明らかにする。

「証明」＝「プログラム」＝「計算」

「計算」「証明」「プログラム」という三つの言葉は、もともと別の意味を持っている。それは今でも同じだ。そして長い歴史を持っている。

この三つの言葉の中で、「計算」の起源が一番古い。4000年以上前の古代バビロニアでは、2の平方根の値も知っていたし、ピタゴラスの定理で直角三角形の斜辺の長さを計算する「計算表」も持っていた。「証明」の概念の起源は、2300年前のユークリッドに帰してもいいと思う。

この中で、「プログラム」という項が一番新しい。その起源は、コンピュータが登場した1950年代を遡ることはないのは明らかだ。

バビロニアの数学　粘土板 YBC 7289

バビロニアでは、2の平方根の値を知っていた。

その近似値は60進法で4桁、
10進法では約6桁に相当する。
対角線に刻まれた数字は、
1, 24, 51, 10。60進では、

$$1 + 24/60 + 51/60^2 + 10/60^3 = 1.41421296...$$

となる。

$$\sqrt{2} = 1.41421296...$$

バビロニアの数学　Plimpton 322

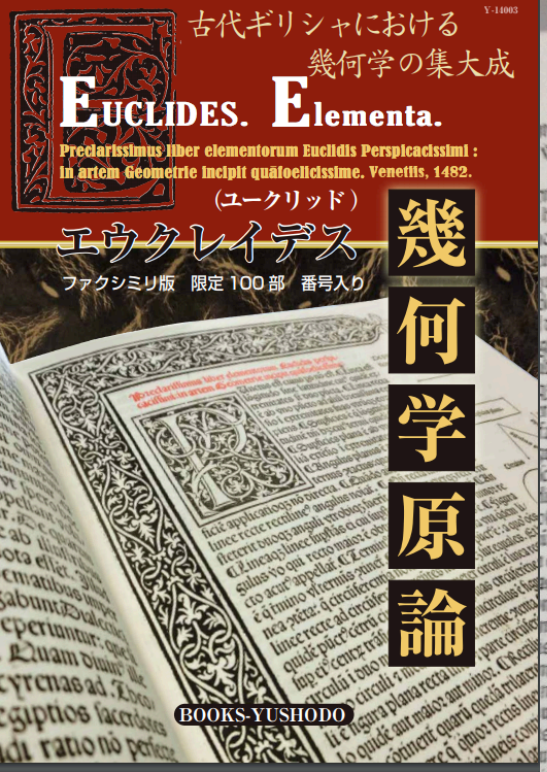
バビロニアでは、ピタゴラスの定理を知っていた。

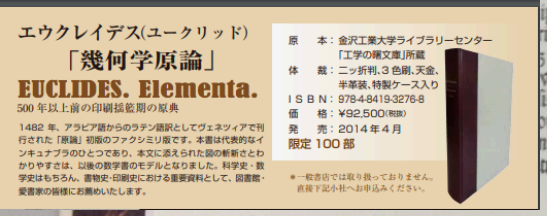
計算

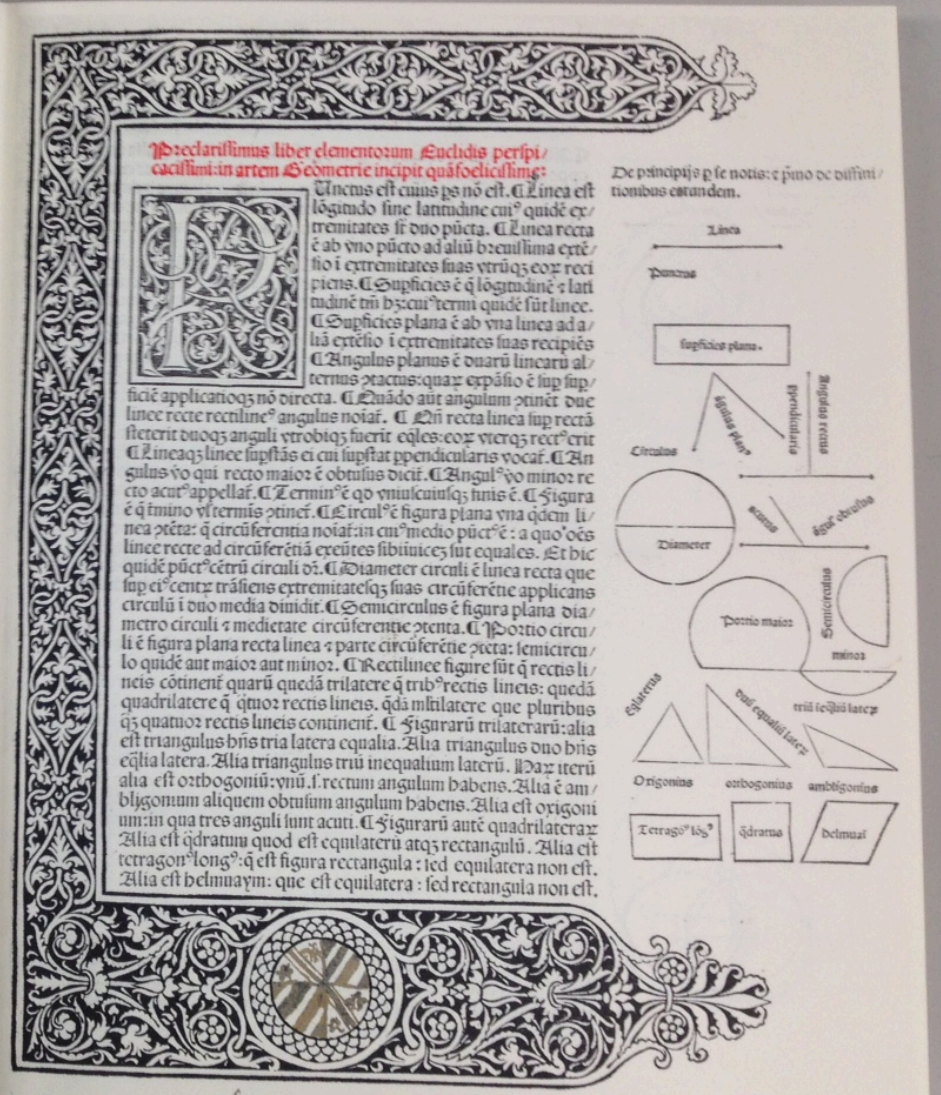
1822 BC ~ 1762 BC

証明

ラファエロの壁画「アテナイの学堂」から



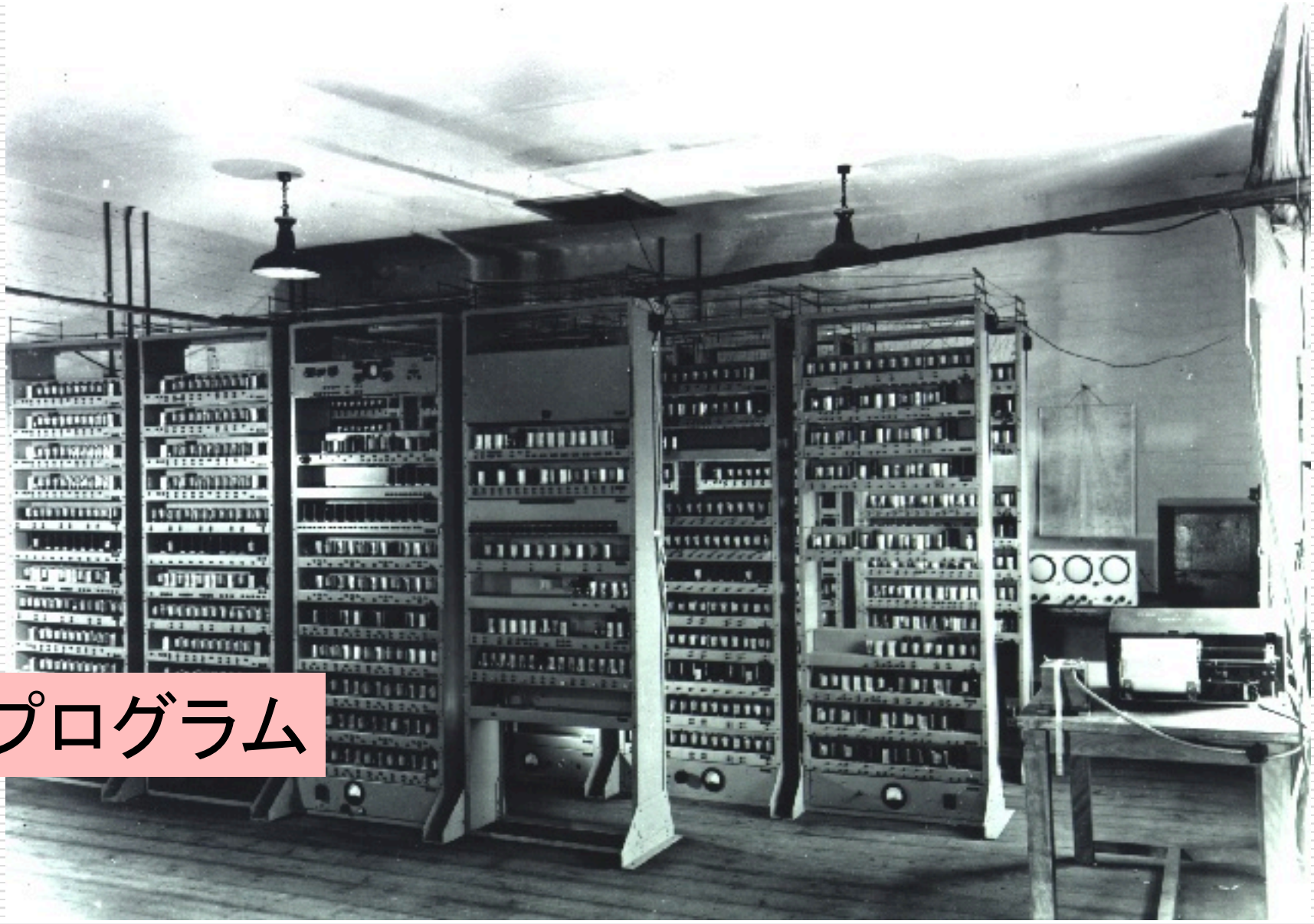




証明

1482 年、アラビア語からのラテン語訳としてヴェネツィアで刊行された「原論」初版のファクシミリ版

EDSAC (Electronic delay storage automatic calculator) 1949



プログラム

amazon web services™

Microsoft Azure

Kubernetes

κυβερνήτης: *Greek for "pilot" or "helmsman of a ship"*
the open source cluster manager from Google

プログラム

「証明」＝「プログラム」＝「計算」

ここでは、機械の能力を人間がどう認識してきたかを、この三つの概念の関係の認識の歴史をメイン・ストーリーにして述べたものである。

概略を述べておこう。

まず、「計算」と「証明」が、本質的には同じものであるという認識が生まれる。1930年代のことだ。

それから40年ほど経って、「プログラム」は「証明」とみなせるという認識が生まれる。1970年代のことだ。

理論的には、この二つの発見で、三項の関係は明確になった。「コンピュータは、プログラムの実行という形で、証明を計算する」のである。

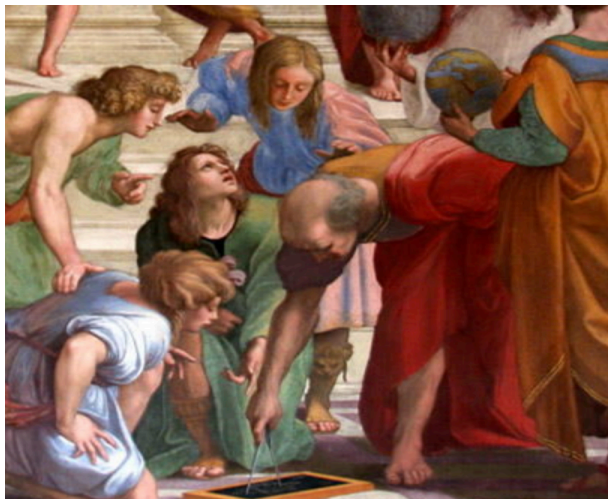
プログラム

1970年代

コンピュータは、
プログラムの
実行という形で、
証明を計算する

計算

証明



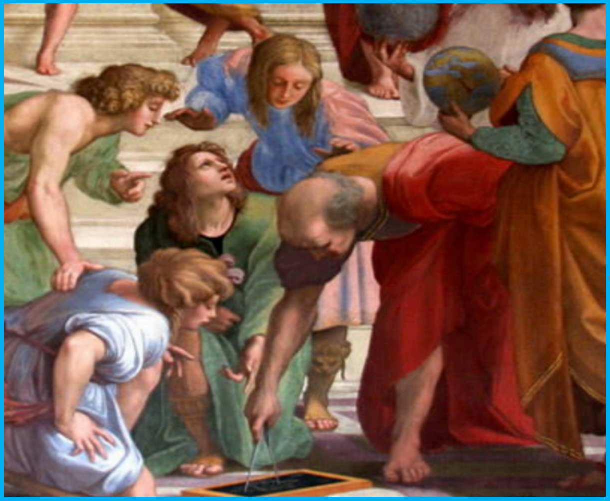
1930年代

「証明」＝「プログラム」＝「計算」

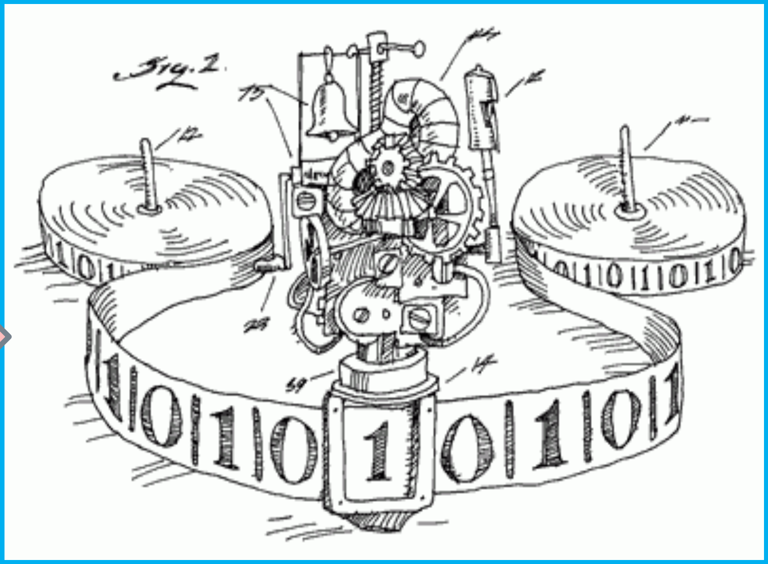
- 理論的には、現在は、三つ目の大きな転換点を迎えている。
- 数学とその証明をコンピュータ・プログラムとして記述しようというVoevodskyのUniMath, プログラムの正しさを数学的に証明しようというChlipalaらのDeep Specificationの動きが、それである。
- 機械が論理的＝数学的推論能力を持つという認識は、現実を動かす力になろうとしていると僕は考えている。

「証明」＝「計算」

証明



計算



「証明」＝「計算」

「証明」＝「計算」の認識の突破口は、今から90年前、ゲーデルの有名な「不完全性定理」によって切り開かれた。

1930年　ゲーデルの不完全性定理

ゲーデルの不完全性定理とは、次のような定理である。

ある形式的体系Sを作ったとしよう。その体系では、初等数論を定義でき、矛盾がないとする。その時、Sに属する命題Gで、Sの中では証明も反証もできないような命題Gが存在する。

Fを、初等数論を含む、無矛盾の形式的体系としよう。この時、この形式的体系Fの無矛盾性は、この体系Fの中では証明出来ない。

「計算とはなにか？」「証明とは何か？」

歴史的に見ると、不完全性定理は、証明あるいは計算の性質についての探求の中で生まれた「最後の結論」ではないのだ。事実は、その逆である。

1930年代に、この分野は、不完全性定理に強烈な刺激を受けて、「証明とは何か？」「計算とは何か？」という基本的な「問い」に突き進むことで発展する。

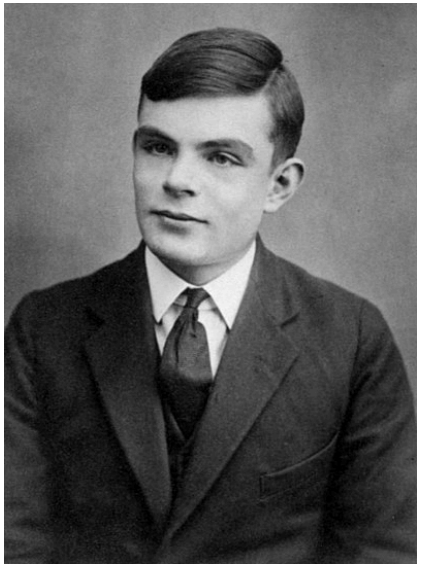
ゲーデルの結果を受けて、「計算可能性」「証明可能性」についての探求が一斉に始まる。

この時代の白眉は、「計算可能性」についての見かけは全く異なるこれらのアプローチが（もちろん、それは、ゲーデルの結果を再現できるものだ）、次々と、「同値」であることが証明されていくことだ！

様々な「計算可能性」へのアプローチと
その同値性の認識

Kurt Gödel
1906-1978

Alonzo Church
1903-1995



Alan Turing
1912-1954

帰納関数論

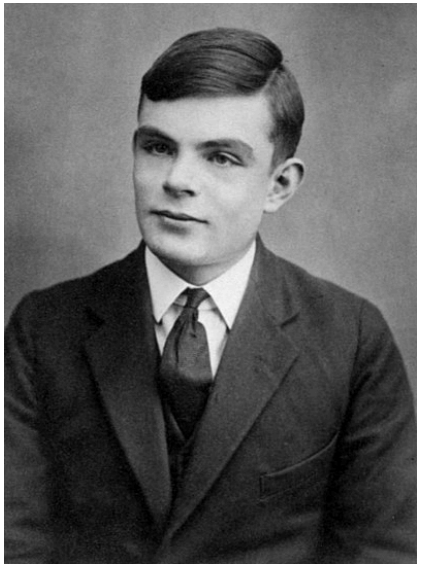
ラムダ計算

チューリング
マシン

様々な「計算可能性」へのアプローチと
その同値性の認識

Kurt Gödel
1906-1978

Alonzo Church
1903-1995



Alan Turing
1912-1954

帰納関数論 ＝ ラムダ計算 ＝ チューリングマシン

チャーチのラムダ計算

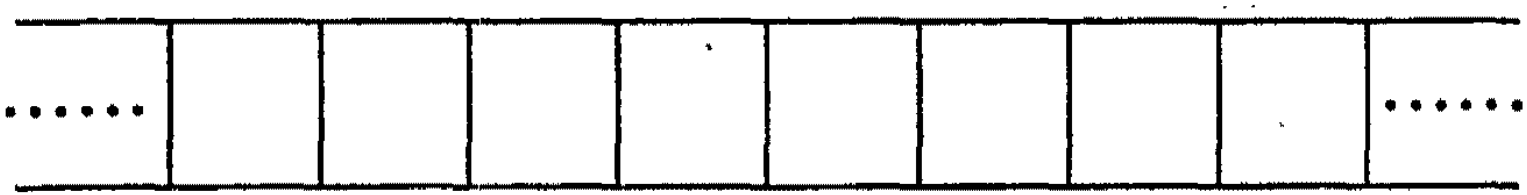
型のないラムダ計算

1. 変数x,...はλ式である。
2. tがλ式で、xが変数であるなら、λxによる抽象化λx.t はλ式である。(abstraction)
3. t, sがλ式であるなら、tへのsの適用 t s は、λ式である。(application)

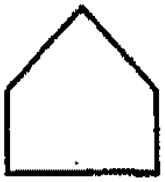
型付きラムダ計算

1. 単純な変数 v_i は型を持つ。 $v_i : \tau$
2. $x : \sigma$ で $e : \tau$ なら、$(\lambda x_\sigma.e) : (\sigma \rightarrow \tau)$
3. $e_1 : (\sigma \rightarrow \tau)$で、$e_2 : \sigma$ なら、$(e_1\ e_2) : \tau$

チューリング・マシン



テープ（マスで区切られている）



ヘッド

テープ：ひとマスずつに区切られており、左右に移動する。長い。

ヘッド：テープのひとマスの記号を読み取り、マシンの「状態」に応じて、次の動作をする。

可能な動作：

1. マス目に記号を書き込む（消す＋書く）。あるいは、そのままにして何も書き込まない。
2. マシンの状態を変える。あるいは状態を同じに保つ。
3. ヘッドを、右あるいは左に移動する。（テープが動く）

チャーチ＝チューリングのテーゼ

1940年代には、今日、「チャーチ＝チューリングのテーゼ」と呼ばれる「計算可能性」についての認識は、確立されることになる。（これが、「複雑性理論」の第一世代である。）

「チャーチ＝チューリングのテーゼ」というのは、次のような提案である。

Every effectively calculable function (effectively decidable predicate) is general recursive

すべての実効的に計算可能な関数（実効的に決定可能な述語）は、一般帰納的である。

「証明可能性」と「計算可能性」の同一性の認識

ゲーデルの不完全性定理から、チャーチ＝チューリング・テーゼへの流れは、今から 90年前のこの時期に、「証明可能性」と「計算可能性」の同一性の認識は、明確に認識されていたことを示す。

しかし、コンピュータは、まだ存在しない！

プログラム

証明

「証明」＝「プログラム」

「カリー・ハワード対応」の発見と「従属型理論」の成立

ゲーデルの発見から40年後の1970年代、論理学・数学の分野で、重要な動きがあった。ハワードの「カリー・ハワード対応」の（再）発見と、それに基づくマーティン・レフの「従属型理論」の成立である。

Curry-Howard対応

- 1940年代のChurchの仕事から、ラムダ計算に対する関心が、ふたたび活発化するのは、20年近くたった1960年代からである。
- 理由ははっきりしている。コンピュータが現実に動き出したからである。
- この時期の代表的な成果は、HowardのCurry-Howard対応の研究である。

Curryの発見
型付きラムダ計算と直観主義論理との対応

- 既に1934年に、Curryは、型付きラムダ計算と直観主義論理とのあいだに、対応関係があることを発見していた。

- 型を持つラムダ計算の関数の型を示す矢印 → と、論理式“A → B” の中に出てくる含意を意味する矢印 → との間に、対応関係があるというのだ。

Curry, Haskell (1934), "Functionality in Combinatory Logic”
http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1076489/pdf/pnas01751-0022.pdf

カリーが気づいたこと
次の二つのルールの形が似ているということ

論理式の推論ルール

$$\frac{A \to B \qquad A}{B}$$

A → B が成り立ち、
Aが成り立つなら
Bが成り立つ

関数の適用の型のルール

$$\frac{f\colon A \to B \qquad a\colon A}{f\ a\colon B}$$

f が A → B という型を持ち、
a がAという型を持つなら
f a は型 B を持つ

論理式の含意を意味する矢印 → と関数の型を示す矢印 →の間に、対応関係があるということ。

ハワードとマーティン・レフが、考えたこと

論理式 A → B が、A → B という関数と同じ型を持つと考えられるとしたら、他の論理式も、型を持つと考えることができるのでは？

例えば、論理式 A ∨ B は、型 A ∨ B を、論理式 A ∧ B は、型 A ∧ B を持つと考えることはできないか？

そして、この論理式（命題：Proposition）を型とみなすアイデアは、うまく機能するのである。こうした考えを、

Proposition as Type

と呼ぶ。

命題Pと証明p ⇨ p : P

彼らは、この時、型Pの項pにあたるものを、命題Pの証明と考えることができることに気づく。それは、命題の証明が何から構成されるかについて、次のような自然な解釈を与える。

- **p: A ∧ B**　　pは、Aの証明とBの証明の両方
- **p: A ∨ B**　　pは、Aの証明、あるいは、Bの証明
- **p: A → B**　　pは、Aの証明からBの証明を導く方法
- **p: (∀x)B(x)**　　pは、任意のaに対してB(a)の証明を与える方法
- **p: (∃x)B(x)**　　pは、あるaに対するB(a)の証明

こうした考えを、

Proof as Term

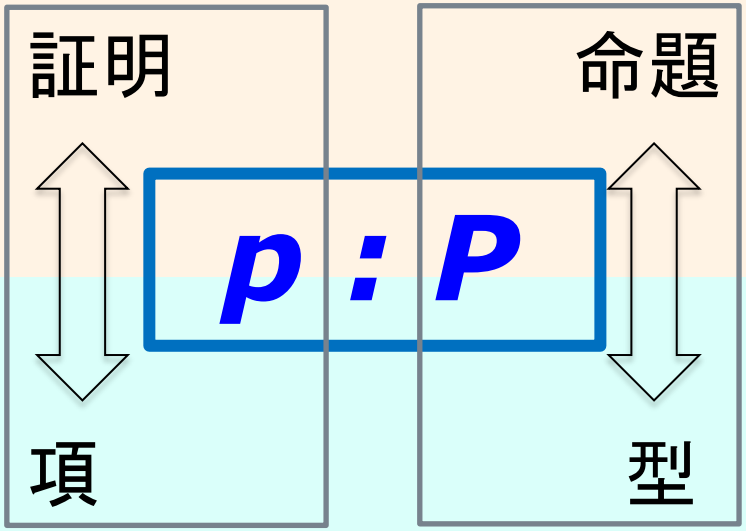
と呼ぶ。

こうした、「型」と「命題」、「項」と「証明」との対応を
Curry-Howard対応 という

「*p* は、命題 ***P*** の証明である」

Proof as Term

証明は
項である



Proposition as Type

命題は
型である

「*p* は、型 ***P*** の項である」

「証明」＝「プログラム」

重要なことは、このCurry-Howard対応は、項の計算をプログラムと見なせば、”Proof as Program”としても解釈出来るということである。

いまから 40年近く前に、論理学者と計算機学者の一部は、「証明」＝「プログラム」という認識にたどり着いていた。

タイムラグ

- 40年に一度の大発見でも、それが他の分野に伝わるには、10年単位の時間がかかることがある。論理学・数学の世界での新しい認識がコンピュータの世界に広い影響を与えるまでには、一定の時間がかかる。
- 30年代のチャーチの「型なしラムダ計算」は、LISP (1958)の理論的基礎。同じチャーチの「型付きラムダ計算」を関数型言語Haskell(1985)は基礎にしている。「従属型理論」に基づくCoqの開発が始まったのは、1984年である。
- ちなみに、マーティン・レフの理論から40年近く経ってからHomotopy Type Theoryで論理学の世界を一新した数学者のVoevdskyは、たまたまコンピュータ実習室で学生の演習を見ていて、「従属型理論」に出会ったという、

プログラム

計算

「プログラム」＝「計算」

「プログラム」＝「計算」

- 「証明」＝「プログラム」＝「計算」というテーゼの中では、「プログラム」＝「計算」というのが、一番わかりやすいように思う。IT技術者なら、「プログラム」＝「アルゴリズム」＝「計算」と考えればいいと思う。
- 「プログラム」という概念自体、20世紀の中頃に生まれたものであるので、。「プログラム」＝「計算」 というテーゼは、三つのテーゼの中では一番新しいものである。
- この認識は、コンピュータの普及と、ITビジネスがコンシューマ中心にシフトして成功する中で、多くの人の中で自然に拡大した。

ただ、次のような問題もある。

「プログラム」＝「計算」

- 現代では、誰もが「コンピュータはプログラムで動く」ことを知っている。ただ、コンピュータは、万能の「情報処理機械」とみなされていて、それが「計算する機械」であることを意識する人は少ない。
- インターネットにしろYoutubeにしろFacebookにしろ、そのサービスを実現するソフトウェアが存在することを、多くの人は知っている。ただ、その基礎が、足し算や掛け算、電卓と同じ「計算」であることに、思いはいたらない。
- サービスを開発するプログラマ自身、数値計算でもしない限り、自分たちの仕事が「計算」の領域に属するという意識は薄いように思う。

サービス

プログラム

計算

証明

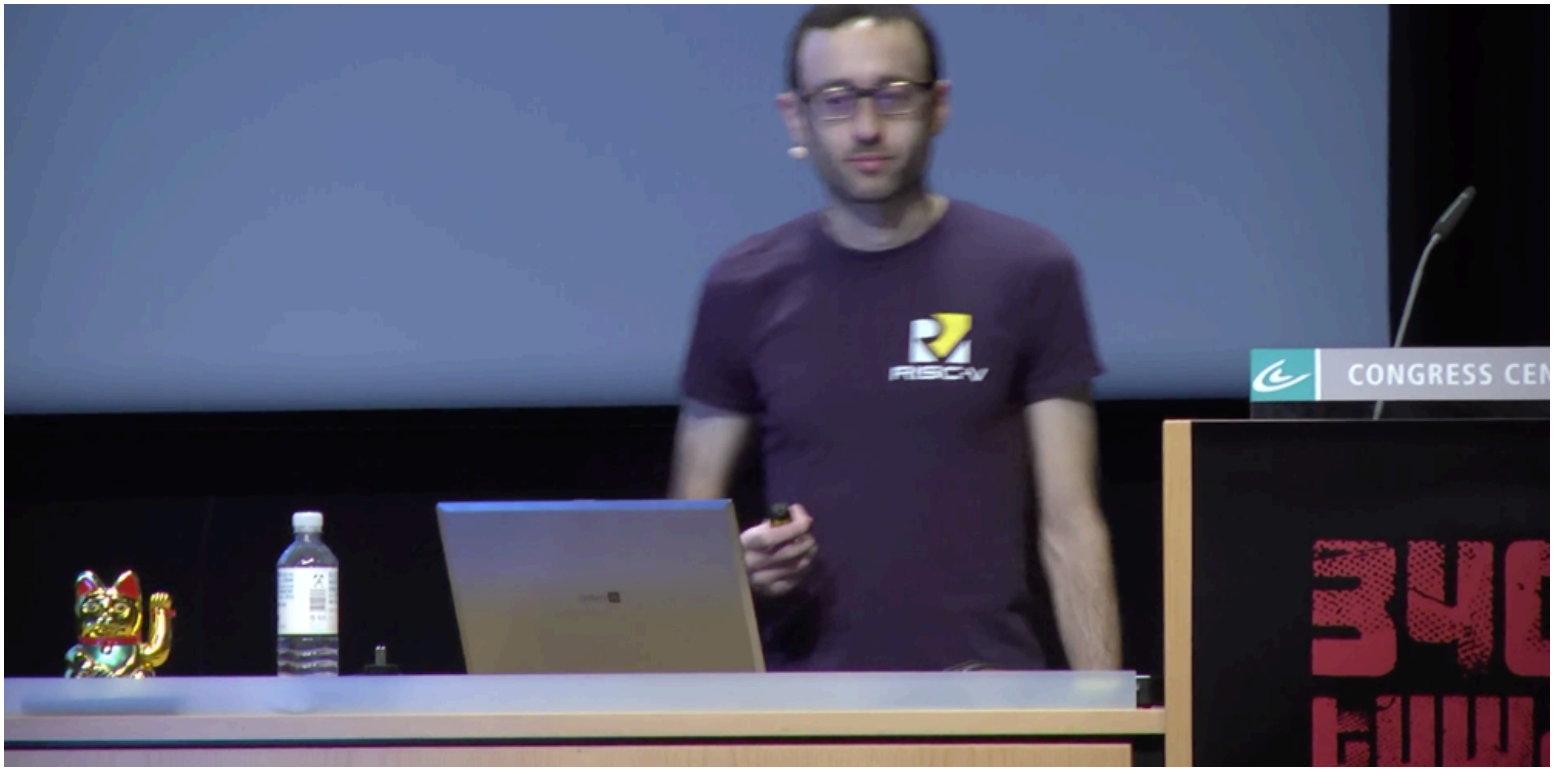
「プログラム」＝「計算」

- 「チャーチ＝チューリングのテーゼ」や「カリー＝ハワード対応」に言及することなく、コンピュータの「論理的＝数学的推論能力」を理解することは可能であろうか？
- 僕は、「プログラムの行っていることは、基本的には、計算である」ということが腑に落ちれば、十分可能だと思う。
- ラムダ計算もチューリング・マシンも、人間が行う計算の、プリミティブだが忠実なモデルに他ならない。それは、コンピュータが行なっている計算と同じものだ。計算に関して言えば、人間とコンピュータに違いはない。
- そうしたプリミティブな機械の働きが、「証明」とつながることを実感するには、いまなら、Coqの経験は、とても役に立つ。
- それは、「計算」＝「証明」、「証明」＝「プログラム」という、多くの人にとっての「ミッシング・リング」を埋めてくれるだろう。
  僕が、IT技術者にCoqの学習を勧める理由は、そこにある。

機械が論理的＝数学的推論能力を持つという認識が、やがて、現実を動かすようになる

「もうじき来るぞ。毎日のソフトとハードの開発に、機械がチェックする数学的証明が」

"Coming Soon: Machine-Checked Mathematical Proofs in Everyday Software and Hardware Developme"



Adam Chlipala http://bit.ly/2GppRxe

VoevodskyとUniMath

- 一昨年（2017年）亡くなったVoevodskyは、Milner予想、Bloch-Kato予想を解くなど、代数幾何でグロタンディックが進もうとした道で、大きな業績を残した。また、Homotopy Type Theory という新しい型の理論で、数学の新しいスタイルでの基礎付けに、画期的な道を開いた。
- Voevodskyの最後の仕事は、数学の基礎とコンピュータに関係していた。彼は、数学の証明に、コンピュータを使うべきだと主張した最初の数学者の一人で、また、そのためのコンピュータによる証明支援システムのライブラリーUniMathを開発した。
  GitHub: https://github.com/UniMath/UniMath
- 2016年9月の講演 "UniMath - a library of mathematics formalized in the univalent style" https://goo.gl/3sJr1M

人間と機械の関係を考える

人工知能論を深めるために考えるべきこと

- 人間の「創造性」と機械の力
  -- Coqでの証明は、人間と機械の協力で成り立っている
  -- 人間と機械との共生の未来？
- 機械との対話から、言語の問題を考える
  -- 開発の問題とは、機械に伝わる言語で、機械に正確に「意味」を伝えることの問題である
- 言語的認識と数学的認識
  -- なぜ、人間にとって、ことばによる説明はわかりやすく、数学的な説明は難しいのか？
- 量子機械の登場と人工知能論への影響
  -- 量子超越性と複雑性理論
- 情報の科学の未来

第二部

Coqの世界

第二部「Coqの世界」 概要

- 第二部「Coqの世界」 は、二つの章からなる。
- 最初の「Coqとのはじめての対話」は、Coqに触れたことのない人に、擬似的にCoqの対話的証明構築環境を体験してもらうことを目的にしている。（先日のハンズオン「はじめてのCoq」の導入部分を流用している。ここだけ「ですます」調になっているのは、そのためである。ご容赦願いたい。）
- 次の「Coqによる形式的仕様の記述例」は、実際に、Coqではどのように仕様を記述するのかの紹介である。ひとつは、簡単な電卓ライクなスタックマシンの仕様記述例である。もう一つは、Deep Specification の代表的な取り組みの一つである。ChlipalaのKamiプロジェクトからのサンプルである。

Coqとのはじめての対話

Coqとのはじめての対話

Coqは、対話型の証明支援システムです。対話するのは人間と機械（Coq）です。

人間と機械の対話でどのように証明が進んでいくかについては、おいおい説明するとして、ここではまず、Coqとのはじめての対話を経験して見ましょう。簡単な証明をしてみます。

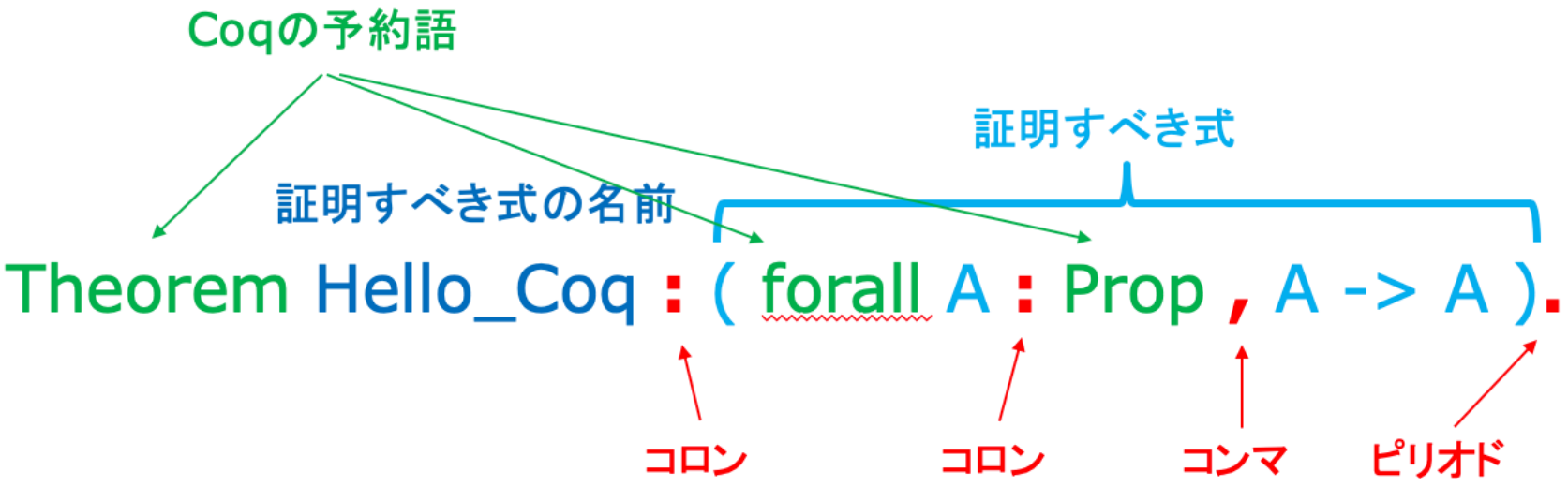
次のブルーの文字列を Coqに入力して実行して見てください。

```
Theorem Hello_Coq : (forall A : Prop, A -> A ).
```

人間の入力

Theorem Hello_Coq : (forall A : Prop, A -> A)

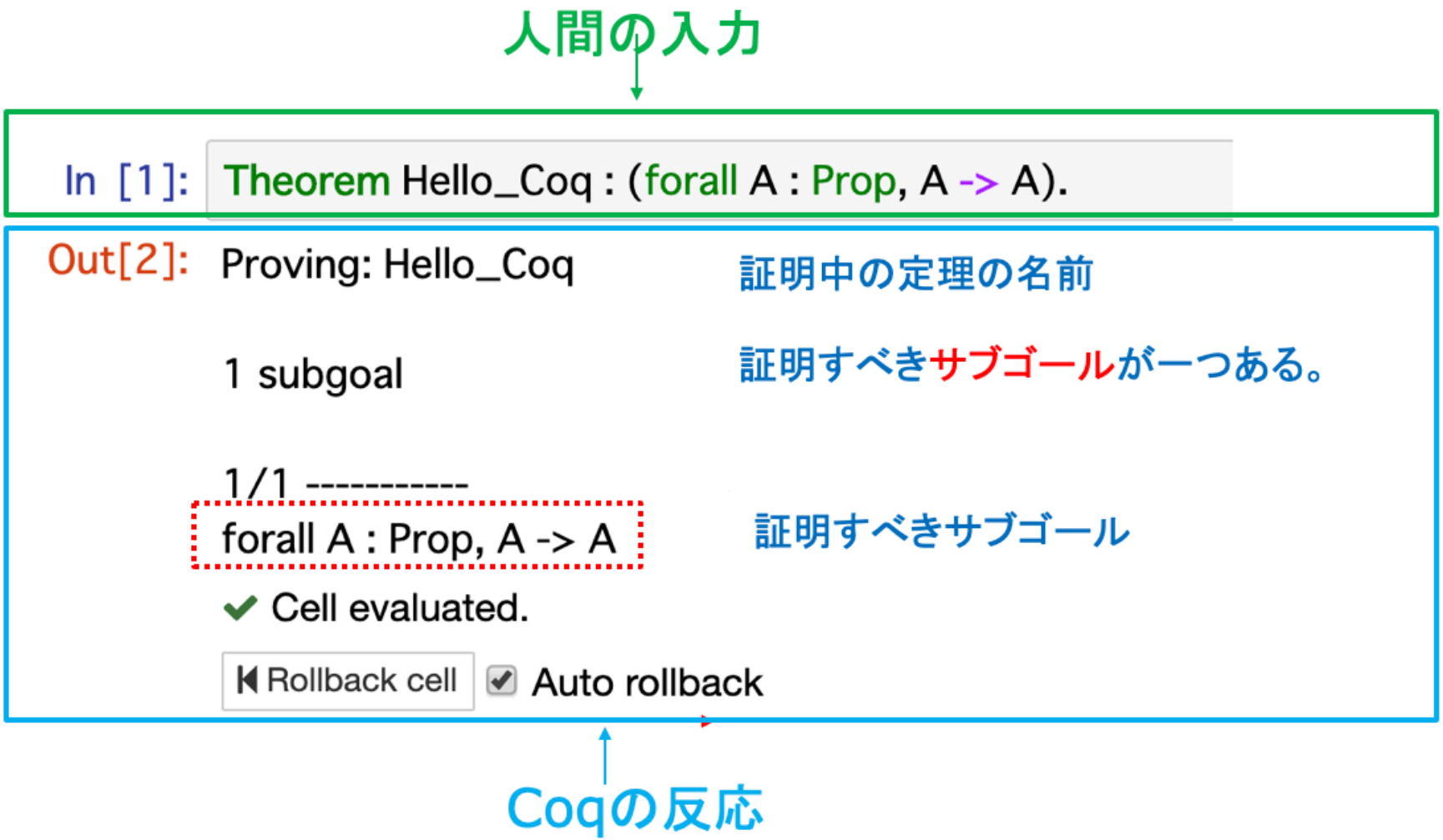
文字列の入力では、コロンやコンマやピリオド、"->"(マイナス記号'-'＋'>')に注意してください。



ここで初めてCoqに証明したい定理の情報を伝えました。

Coqから次のような反応があるはずです。

Coqが返した情報の意味は、次のようなものです。



今度は、次の文字列を Coqに入力して実行して見てください。
(最後の ′.′ ピリオド、忘れないでください。)

```
Proof.
```

次のような反応がcoqからあるはずです。基本的には、先の反応と同じです。 coqコマンドの "Proof."は、これから証明が始まることを宣言しています。

ここでは、先に説明していなかった下二行の説明をしています。これらの出力は、coq-jupyterからのものです。この tutorial では、ツールとして coq-jupyter を利用しています。

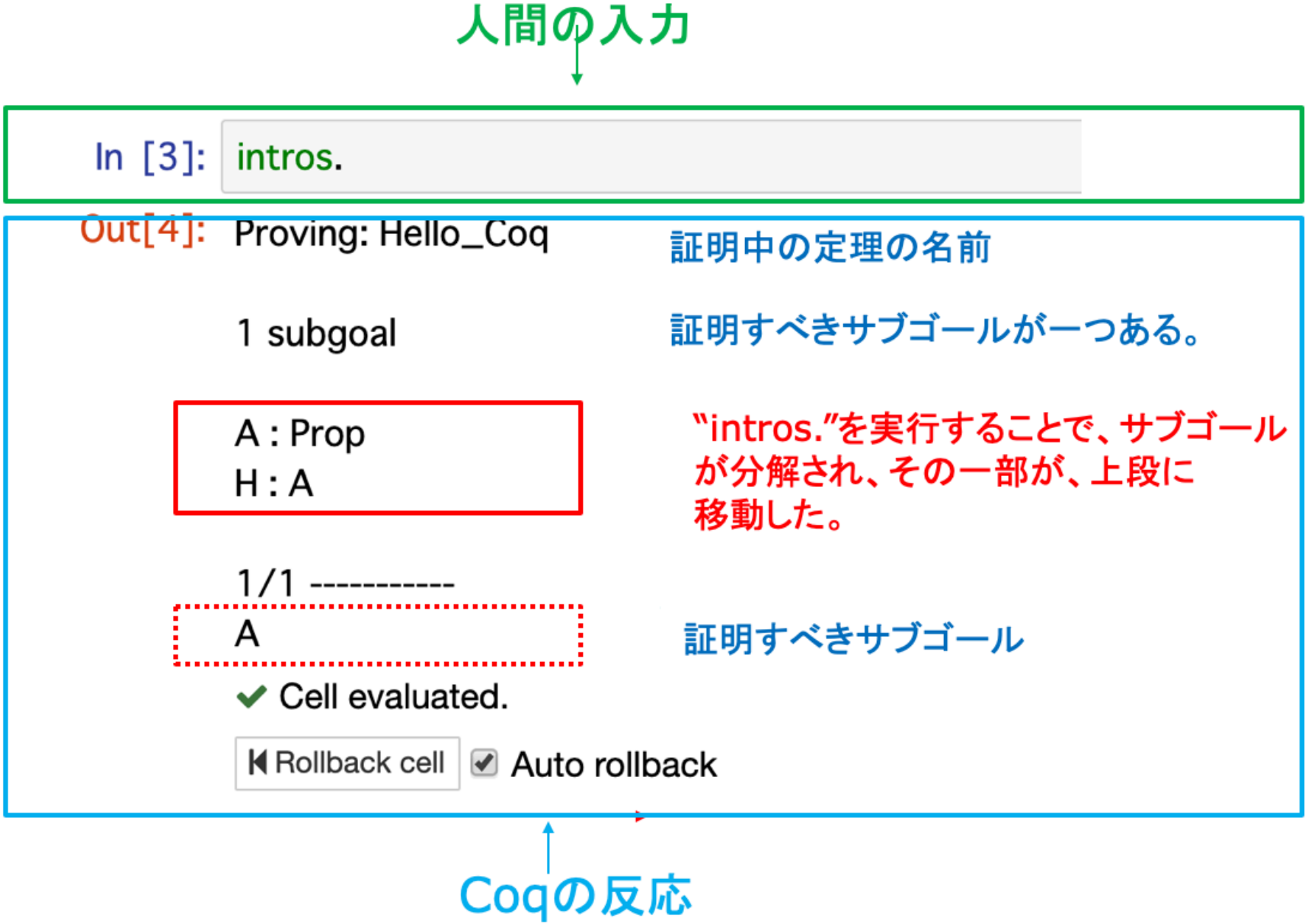
"Cell evaluated"がでていれば、証明は、順調に進んでいると思って構いません。


人間の入力
In [2]:
Proof.
Out[3]:
Proving: Hello_Coq
1 subgoal
1/1 ----------
forall A : Prop, A -> A
Cell evaluated.
Rollback cell
Auto rollback
coq-jupyterの出力
セルが、評価されたことを表す
エラーがあった特にやり直す
Coqの反応

今度は、次の文字列を Coqに入力して実行して見てください。
(最後の ′.′ ピリオド、忘れないでください。)

```
intros.
```

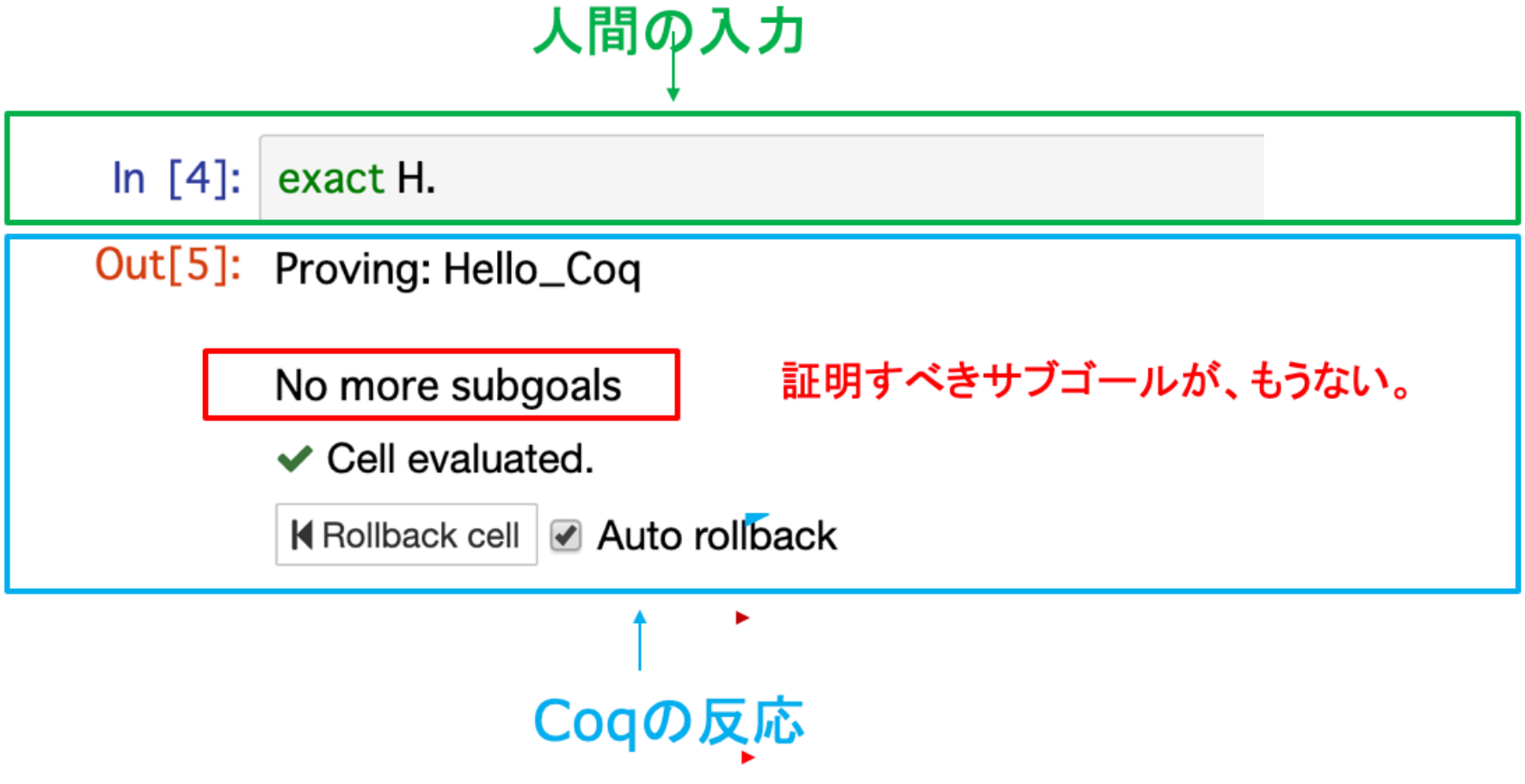
次のような反応があるはずです。その意味を説明します。


人間の入力
In [3]: intros.
Out[4]: Proving: Hello_Coq
証明中の定理の名前
1 subgoal
証明すべきサブゴールが一つある。
A : Prop
H : A
"intros."を実行することで、サブゴールが分解され、その一部が、上段に移動した。
1/1 ----------
A
証明すべきサブゴール
Cell evaluated.
Rollback cell
Auto rollback
Coqの反応

今度は、次の文字列を Coqに入力して実行して見てください。
(最後の ′.′ ピリオド、忘れないでください。)

```
exact H.
```

次のような反応があるはずです。

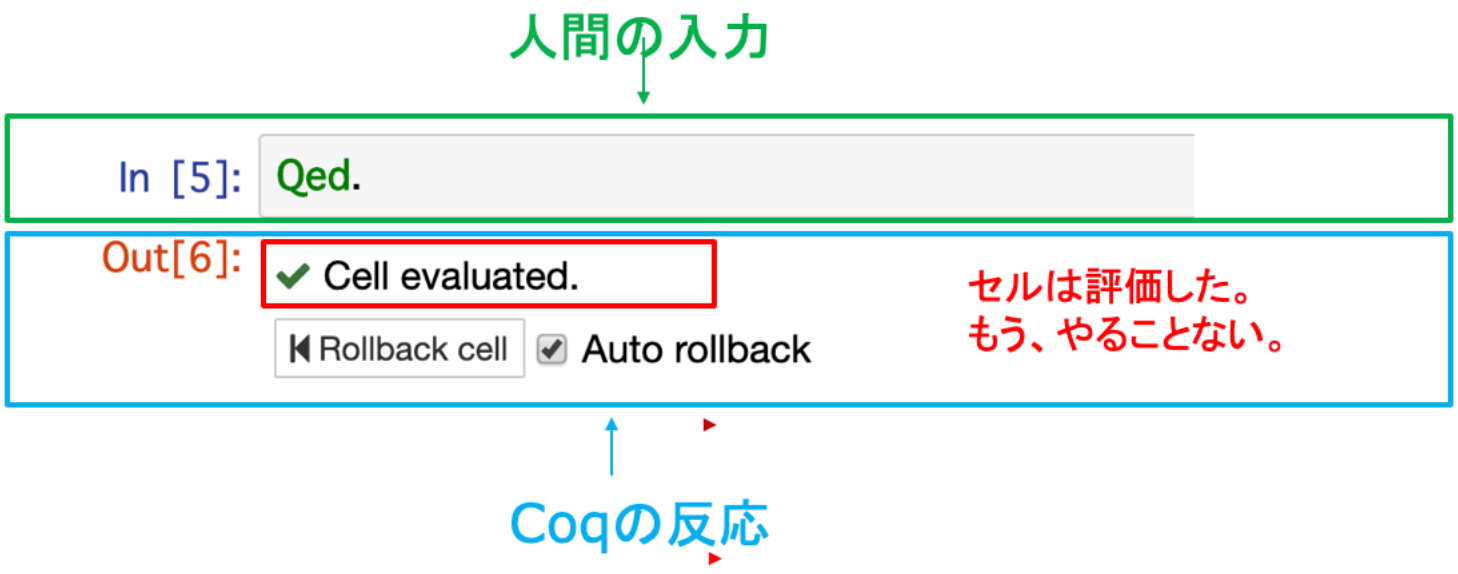

人間の入力
In [4]:
exact H.
Out[5]:
Proving: Hello_Coq
No more subgoals
証明すべきサブゴールが、もうない。
Cell evaluated.
Rollback cell
Auto rollback
Coqの反応

“No more subgoals” というのは、重要なメッセージです。証明すべきサブゴールが残っていないというのは、証明が終わったことを意味します。

証明の終了を伝える、次の文字列を入力してください。

Qed.

次のような反応があるはずです。



これで、証明が終わります。

Coqとのはじめての対話をふりかえる

「証明が終わった」と言われても、あまりピンとこないかもしれません。

それは、今回の「Coqとのはじめての対話」が、「対話」になっていなくて、言われたままの文字列を入力してきたことに原因があります。

今回、人間が行なった入力をまとめておきましょう

```
Theorem Hello_Coq : (forall A : Prop, A -> A ).
Proof.
  intros.
  exact H.
Qed.
```

次のビデオで、今回の「対話」を、もう一度、振り返ってみようと思います。

人間はCoqに何を伝えたか？

Coqとのはじめての対話をふりかえる

前回のビデオで、Coqとのはじめての対話の様子を見てきました。ただ、これが「Coqでの証明だ」と言われても、あまりピンとこなかったかもしれません。

人間が、Coqに何を伝えようとし、
Coqが、人間に何を伝えようとしたのか

を、あらためて振り返って見ましょう。

前回、人間がCoqに伝えたことをまとめてみました、

```
Theorem Hello_Coq : (forall A : Prop, A -> A ).
Proof.
  intros.
  exact H.
Qed.
```

これだけ見ていると、これが定理Hello_Coq : (forall A : Prop, A -> A)の証明とは、とても思えないと思います。

逆に、定理Hello_Coq が与えられた時、上のような証明をすぐに思いつくことは、ほとんどの人にはできないと思います。

Coqの証明は、人間とCoqの「対話」を通じて進行する

それには理由があります。

Coqの証明は、人間とCoqの「対話」を通じて進行します。

人間は、Coqの反応を見て、証明を進めるために次にCoqに何を命ずるかを考えます。

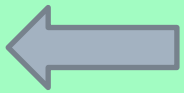
そのやり取りの過程が省略されて、その結果だけを抜き出しても、その命令を選択した意図は見えてきません。

証明を進めるために、人間がCoqに与える命令を **tactic** と呼びます

tactic というのは「戦略」という意味ですね。Coqの反応を見て、証明を進めるために、どのような攻め方をするのかと考えて出てきた結論が、あるtacticを選択するということです。

なぜ、あるtactic が選ばれたかは、tacticの選択の直前のCoqの反応を見てみないとわかりません。この例では、introsや exact H がtacticです。

```
Theorem Hello_Coq : (forall A : Prop, A -> A ).
Proof.
  intros.
  exact H.
Qed.
```



tactics

人間は、Coqとの「対話」を毎回繰り返す必要があるのでしょうか？

「これが定理Hello_Coq (forall A : Prop, A -> A)の証明とは、とても思えない」と、先ほど書きました。

```
Theorem Hello_Coq : (forall A : Prop, A -> A ).
Proof.
  intros.
  exact H.
Qed.
```

ある証明が正しいことを示すために、人間は、Coqとの「対話」を毎回繰り返す必要があるのでしょうか？　そうでは、ありません。

意味不明なのは慣れない人間にとってのこと。Coqは、これを正しく証明として認識し、それが正しい証明かチェックできます。

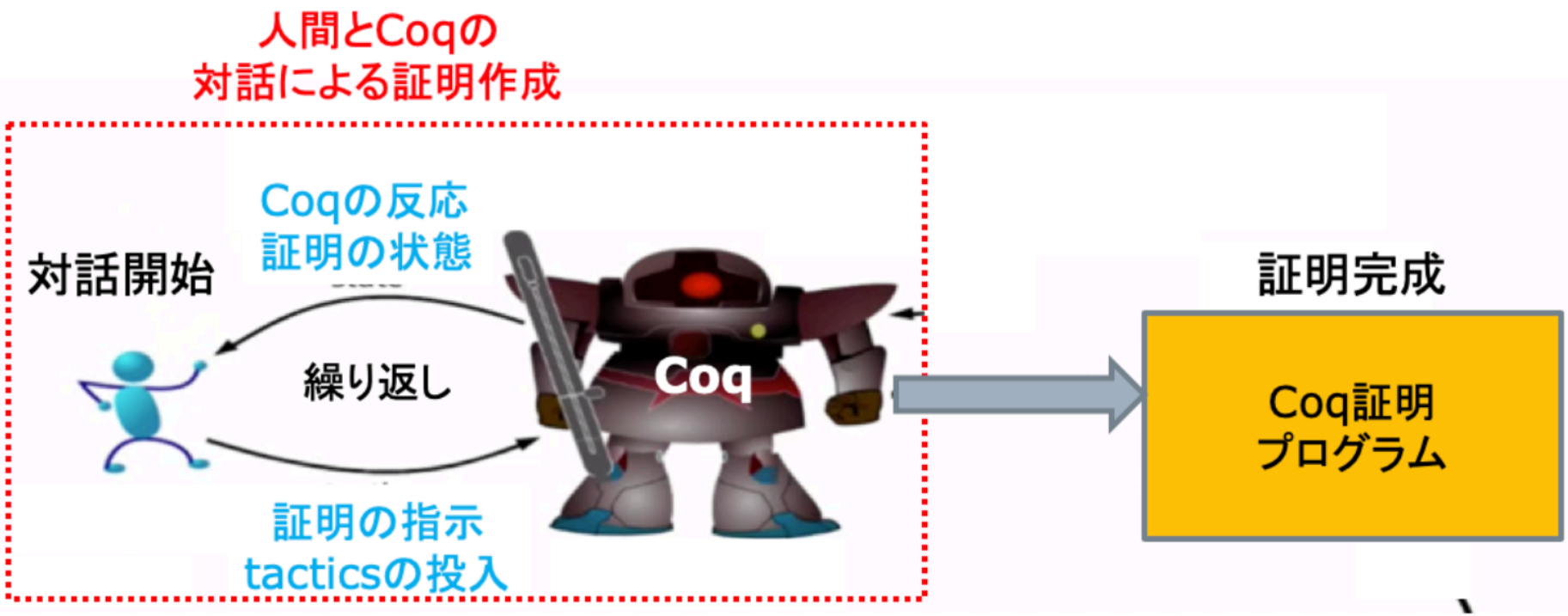
完成した証明を、「対話」なしに、人間はCoqに与えることができます。

- 人間とCoqが「対話」していったん完成した証明は、人間はtactic ごとのCoqの反応を見ることなく、それを丸ごとCoqに与えることで証明を実行することができます。

- これは、証明の完成のためにCoqと「対話」した相手が誰であろうと構わないという点で、実践的には、とても重要なことです。我々は、第三者の行なった証明を、利用することができるのです。

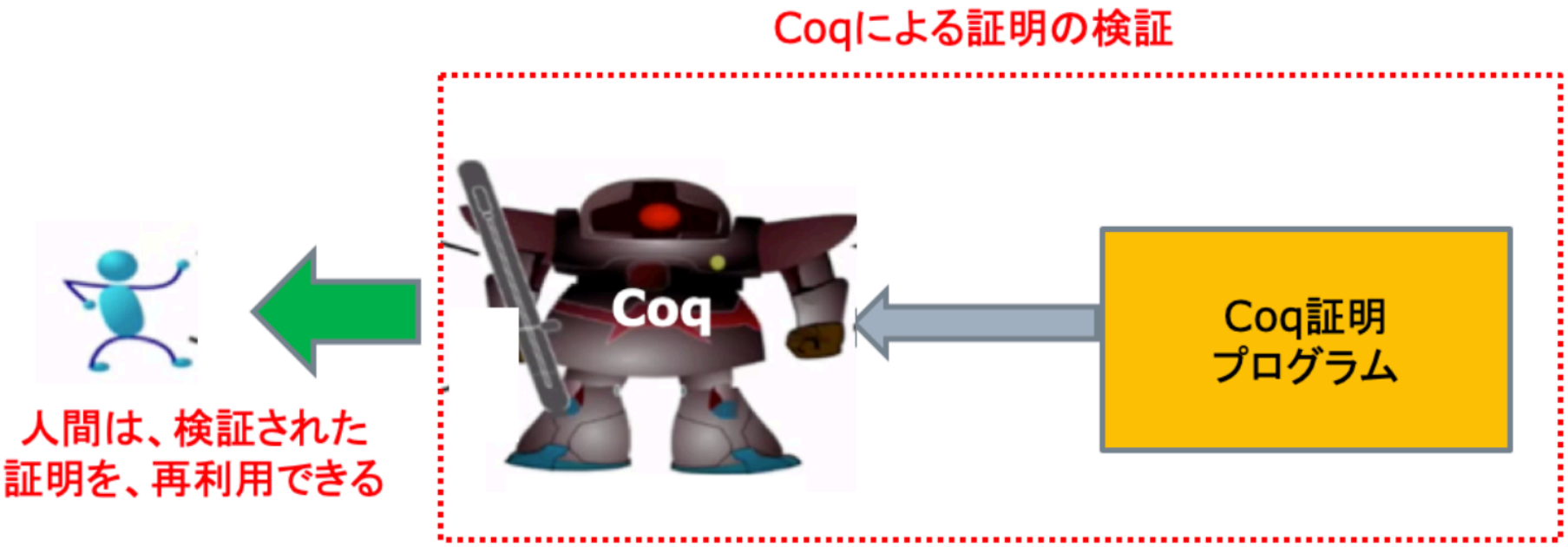
Coq利用の二つのモード

Coqの利用には、二つのモードがあります。
一つは、「対話型証明」モードです。



Coq利用の二つのモード

Coqの利用には、二つのモードがあります。
もう一つは、「証明自動検証」のモードです。



今回のハンズオンの演習問題では、演習問題として実際に自分で解いた証明済の定理のみを使って、証明を行うことを原則として配列されています。

それは、このハンズオンが、Coqでの証明スキルを学ぶことを目的にしているからです。

第一章の最後に、tactic tauto を紹介しました。このtacticは、命題論理の大部分を自動的に証明します。実際には、わざわざ命題論理の命題を証明しなくても、Coqはそれを解いてくれます。それでも基本的な命題論理の証明方法を学ぶことは、Coqで証明を行う上での基本になります。

同時に、「実際に自分で解いた証明済の定理のみを使う」というのは、Coqの実際の応用の場面では、とても強い制限になることに注意してください。

数学は、整合的な「累積的知」の体系です。

ここでは、重要なことが二つあります。一つは、何かを証明するためには、証明済の命題のみに依拠しなければならないということです。「自明のことだろう」で済ますわけには行かないのです。それは、数学的な方法にとっては、本質的に重要なことです。

ただ、もう一つ重要なことは、その命題を、「実際に自分で解く」必要はないということです。誰が証明したかわからなくても、さらには、その証明の詳細を知らなくても、「その定理が正しいものである限り」、我々はその定理を利用できます。

それは、Coqの応用に限らず、数学の応用全般について言えることです。数学は、原理的には、整合的な「累積的知」の体系です。この原理はとても強力なものです。実際の場面では、この原理を利用しない手はないのです。

数学の「整合的な累積的知の体系」という原理と自由な定理利用の背景には、「本当にその定理は正しいのか？」という基本的な疑問が潜在的には存在します。

こうした厄介な問題にたいしても、Coqはとてもスマートな解決方法を提供します。それは、ある定理が、Coqでの証明という形で提供されているなら、その証明が正しいか否かを、我々は簡単に「検証」できるということです。

ソフトウェア・エンジニアリングでの "Deep Specification" や、数学での "UniMath" といった注目すべきムーブメントは、Coqのこうした能力によってドライブされています。

Coqは人間に何を伝えたか？

ふりかえり

最初のビデオ「Coqとのはじめての対話」で、Coqとのはじめての対話の様子を見てきました。

ただ、これが「Coqでの証明だ」と言われても、あまりピンとこなかったかもしれません。

ということで、前回のビデオ「人間はCoqに何を伝えたか？」で、人間が、Coqに何を伝えようとしたかを見てきました。

ここでは、人間は、**tactics** という特殊な言葉で、Coqに話しかけていました。

Coqの証明は、人間とCoqの「対話」を通じて進行します

Coqの証明は、人間とCoqの「対話」を通じて進行します。

人間は、Coqの反応を見て、証明を進めるために次にCoqに何を命ずるかを考えます。

そのやり取りの過程が省略されて、その結果だけを抜き出しても、その命令を選択した意図は見えてきません。

証明を進めるために、人間がCoqに与える命令を **tactic** と呼びます

tactic というのは「戦略」という意味ですね。Coqの反応を見て、証明を進めるために、どのような攻め方をするのかと考えて出てきた結論が、あるtacticを選択するということです。

なぜ、あるtactic が選ばれたかは、tacticの選択の直前のCoqの反応を見てみないとわかりません。

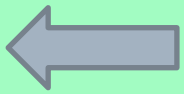
Coqで証明を進めるためには、人間の入力に対して、Coqがどのように反応したか、その意味をよく知る必要があります。

このビデオでは、Coqが人間に何を伝えようとしたを説明します。

Coqは、反応として人間に何を伝えたか？

ここでは、先の例で人間が intros と打ち込んだ時のCoqの反応を、改めて、見ることにしましょう。

```
Theorem Hello_Coq : (forall A : Prop, A -> A ).
Proof.
  intros.
  exact H.
Qed.
```



ここ

Coqは、反応として人間に何を伝えたか？

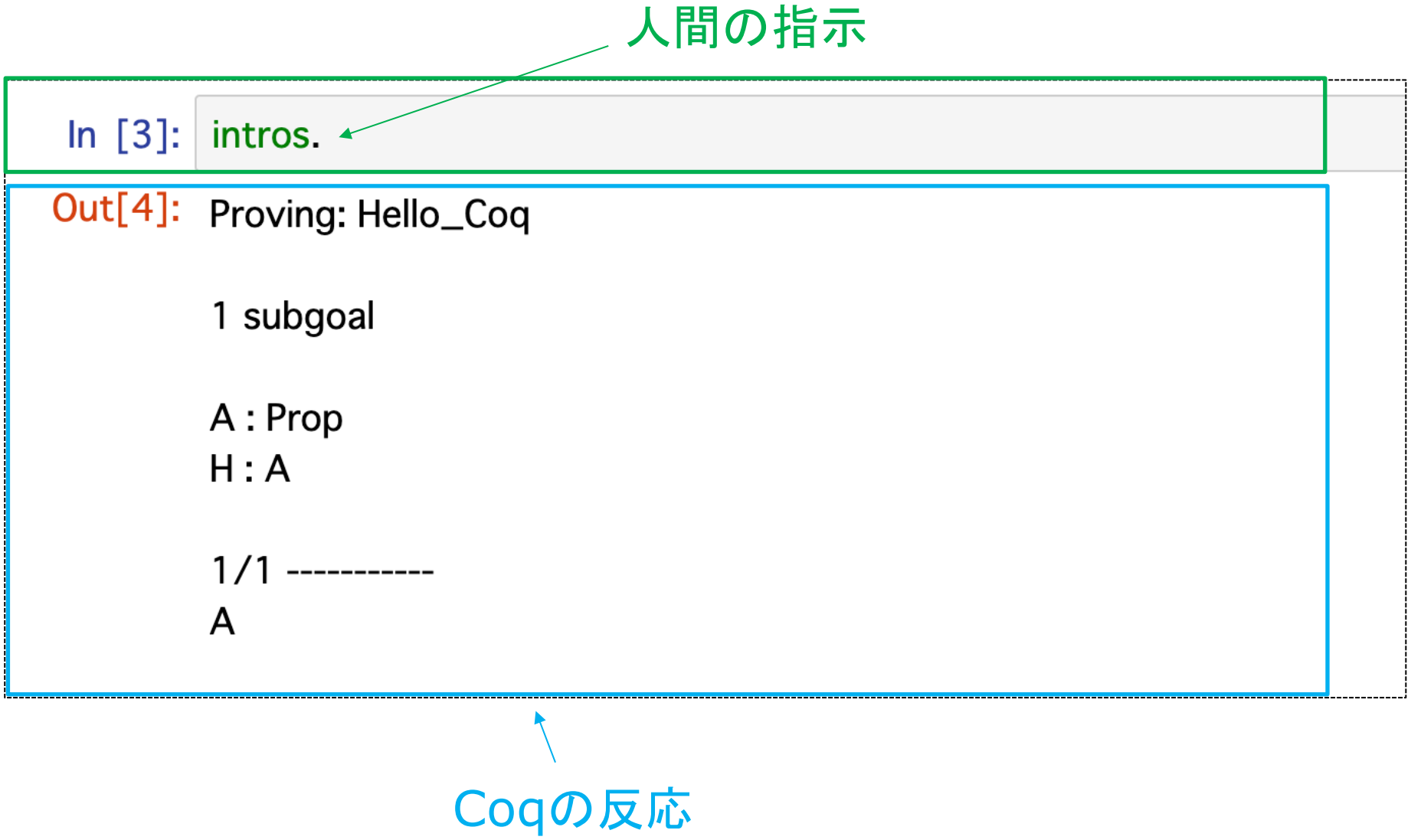
次のような情報が含まれていることがわかります。

- 証明中の定理の名前
- 証明すべきサブゴールの数

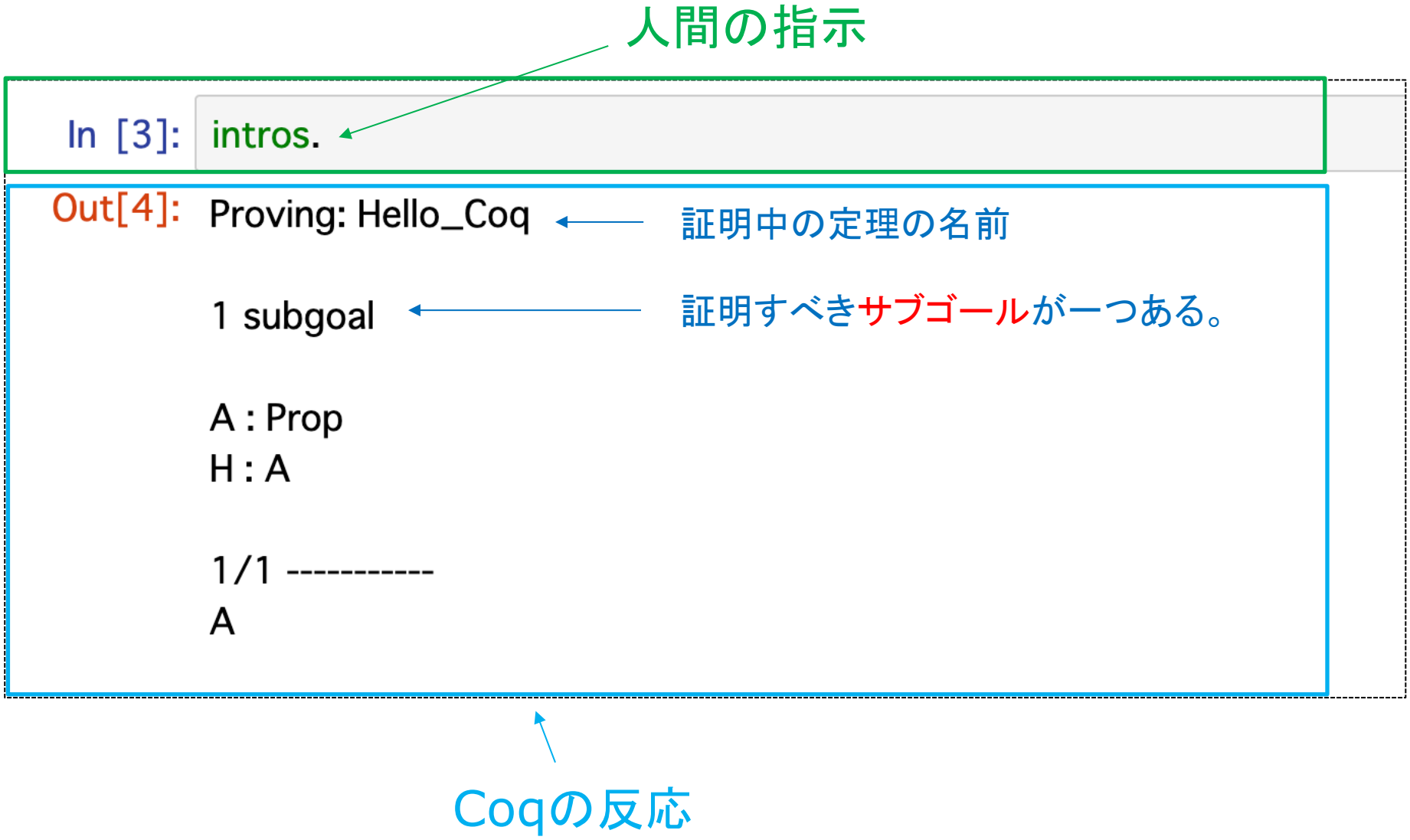
- 証明の状態を表す大事な情報（この情報は、横線の上下で大きく二つに分かれています。）
  - サブゴールを証明するにあたって前提として利用できる仮説
  - 証明すべきサブゴール

Coqは、反応として人間に何を伝えたか？

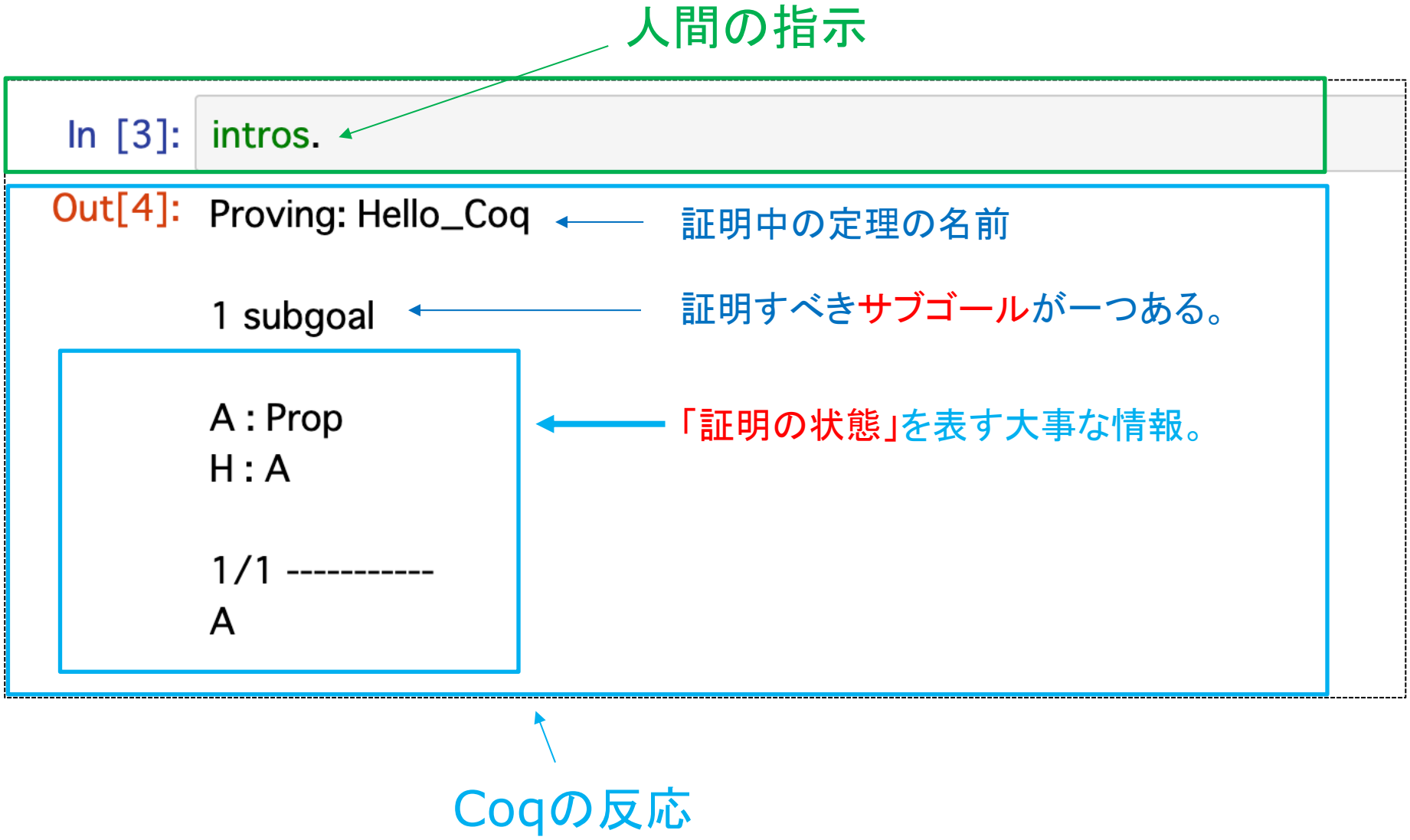
Coqは、反応として人間に何を伝えたか？



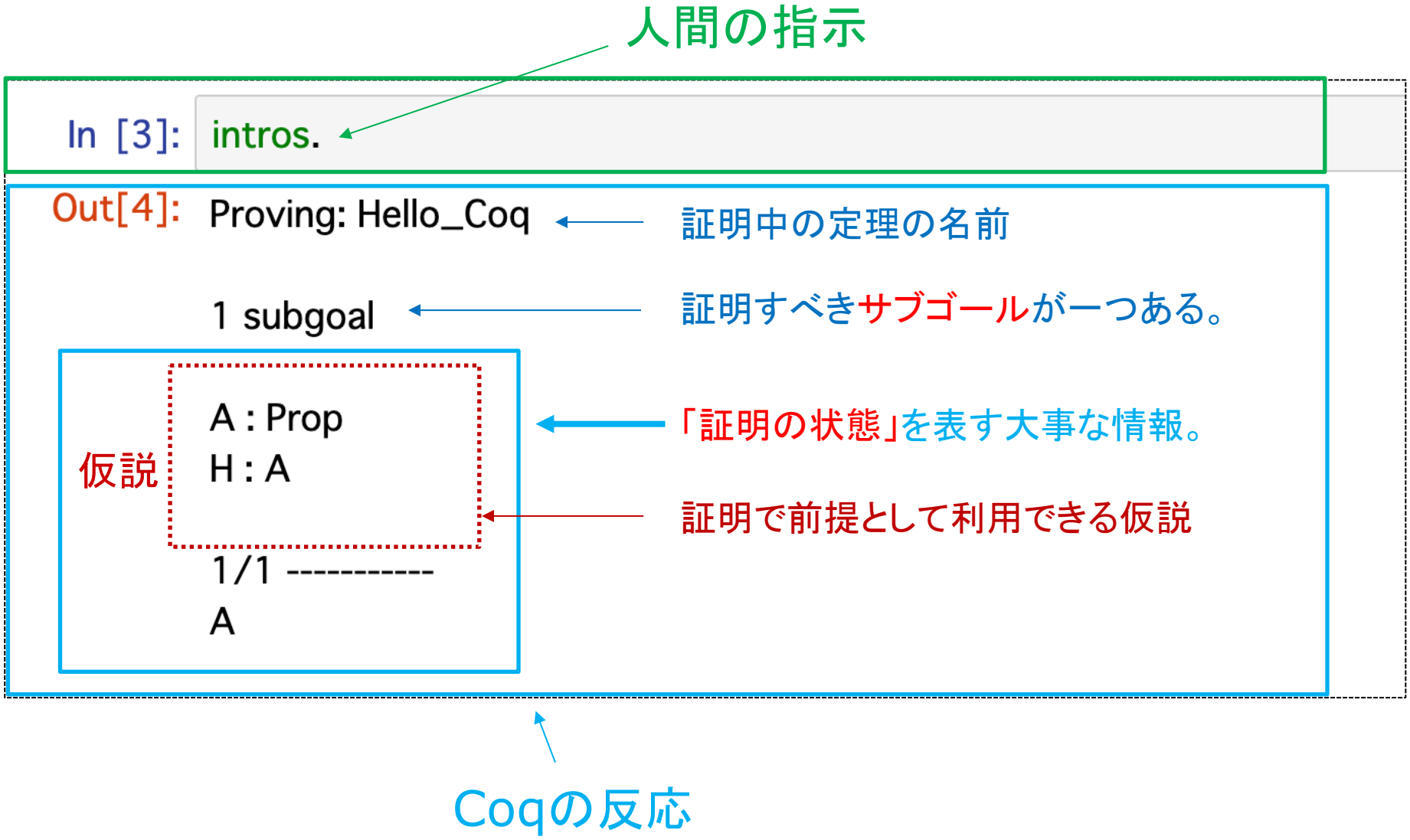
Coqは、反応として人間に何を伝えたか？



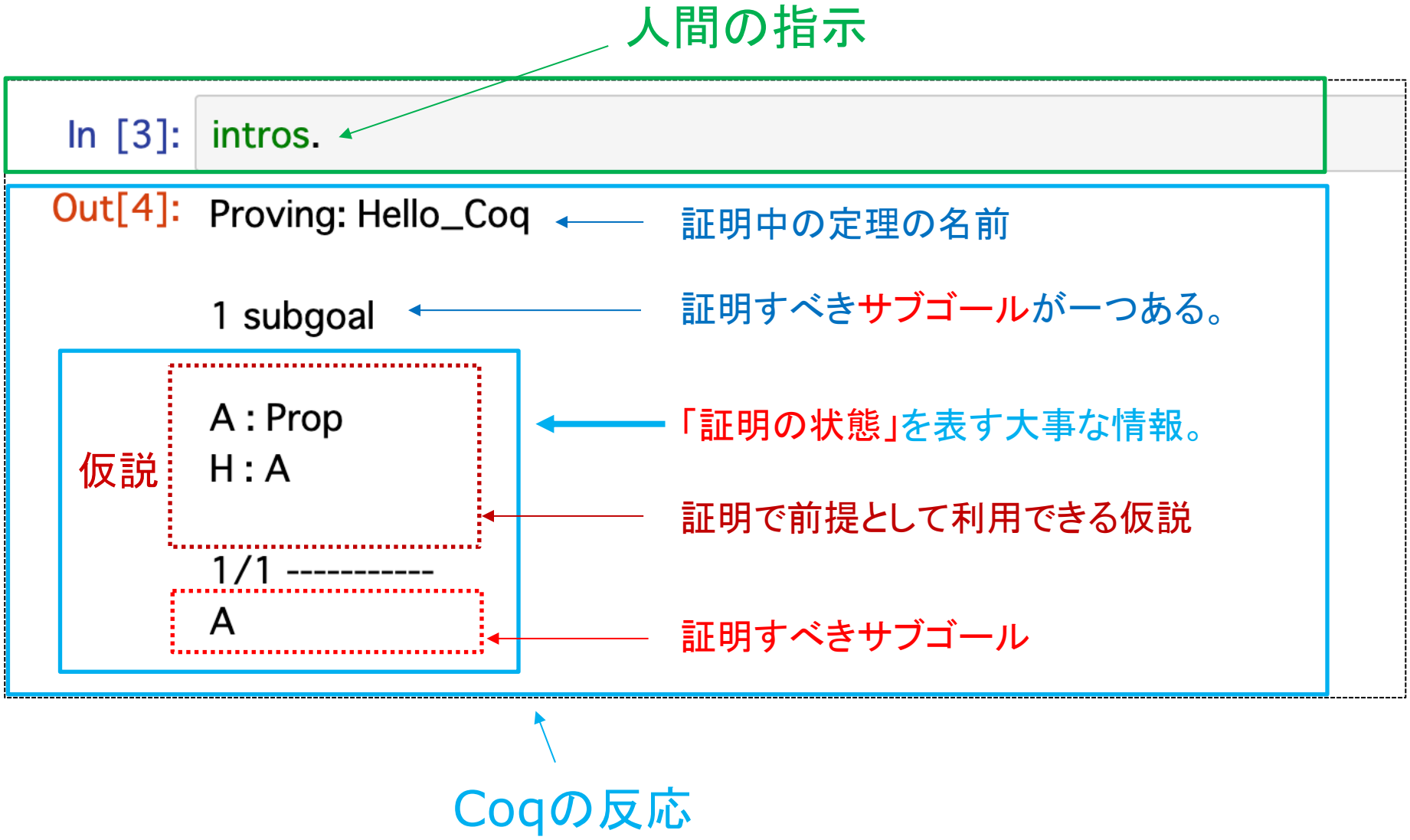
Coqは、反応として人間に何を伝えたか？



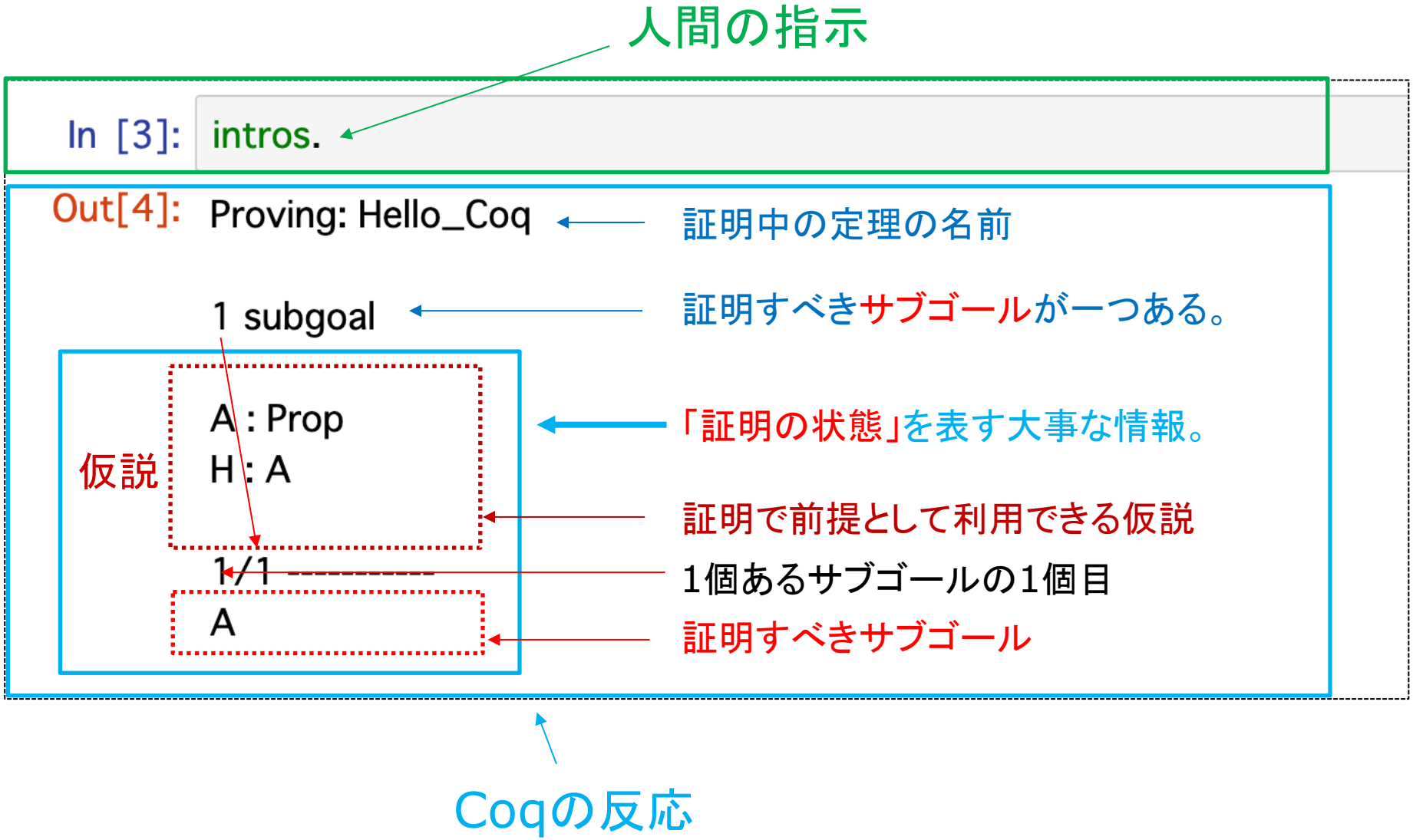
Coqは、反応として人間に何を伝えたか？



Coqは、反応として人間に何を伝えたか？



Coqは、反応として人間に何を伝えたか？



Coqは、その時点での「証明の状態」を返します。
tacticsは、「証明の状態」を変化させます。

Coqが返す反応で一番大事なのは、その時点での「証明の状態」を表す情報です。

その状態は、直前に人間が投入した tactic コマンドによって変化します。

tactics は、まさに、証明の状態を変化させるために、人間が利用するコマンドです。

「証明の状態」には、当面の証明で集中すべき
サブゴールが含まれています。

「証明の状態」の情報のうち、「サブゴール」といわれるものに注目しましょう。

Coqの反応の中に含まれるサブゴールというのは、「当面は、この問題の証明に集中しようよ」という、Coqから人間へのサジェスションです。Coqから人間への指示と思っても構いません。

Coqは、大きな問題を簡単な問題に分割します

人間とCoqは、大きな問題を複数のより簡単な問題に分割して、一つづつ分割された問題を解こうとします。こうして分割された問題の一つが、サブゴールです。

ですので、ある問題を解くために必要なサブゴールの数は、一つとは限りません。

次の例は、destruct というtactic を実行した結果、二つのサブゴールを持つ証明の状態が生まれたことを示しています。

人間の指示　destruct tactic を使っている

```
In [4]: (* A\/B をdestructすると、Aが仮定に残り、サブゴールが二つに分割される。*)
        destruct A_or_B.
```

```
Out[5]: Proving: or_comm

2 subgoals

A, B : Prop
H : A
1/2 ----------
B \/ A
2/2 ----------
B \/ A
```

✔ Cell evaluated.

Rollback cell ☑ Auto rollback

証明すべきサブゴールが2つある。

仮説

証明で前提として利用できる仮説

証明すべきサブゴールの一つ目

証明すべきサブゴールの二つ目

Coqの反応

Coqでの証明の最終目標は、サブゴールをなくすことです。

先に述べたように、Coqは、元の問題をより簡単な問題（サブゴール）に分解して、それをすべて解こうとします。

すべてのサブゴールが解かれたとき、元の問題の証明が終わります。

別の言い方をすれば、Coqでの証明の最終的な目標は、証明されていないサブゴールの数をゼロにすることです。

Coqによる形式的仕様の記述例

Agenda Part II　Coqの世界

- Coqとのはじめての対話
- 人間はCoqに何を伝えたか？
- Coqは人間に何を伝えたか？
- Coqによる形式的仕様の記述例
- 単純なStack Machineの例

単純なStack Machineの例

http://adam.chlipala.net/cpdt/

https://github.com/phlummox/cpdt/blob/master/src/StackMachine.v

やりたいこと

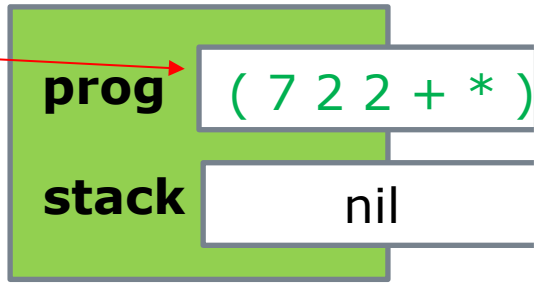
1. (+ ２ ２) あるいは (* (+ ２ 2) ７) のような、自然数の和と積からなる式の値を計算するスタック・マシンの仕様を作る。
   - スタック・マシンは、二つのスタック、プログラムを格納するスタックと、計算作業用のスタックをもつ。
   - 与えられた式は、スタック・マシンのプログラムに変換されて、プログラム・スタックに格納される。
2. この変換 compile が、全ての式について正しく行われることを証明する。
3. このマシンを、等号と不等号を含むように拡張する。

Stack Machine

式 exp
(* (+ 2 2) 7)

compile
(* (+ 2 2) 7)

Stack Machineの実行
(* (+ 2 2) 7)
.
.
.



28

留意すべきこと

- 仕様では、SyntaxとSemanticsの二つを記述する。Semanticsの記述するという表現が、特徴的。
- Denotational Semantics(表示的意味論)とは、状態の変化で意味を表示するものと思っていい。入力を出力に変える関数は、表示的意味論で利用できる。
- ここでのSemanticsの記述は、仕様の関数による実装と、事実上、同じものである。
- この例では、全ての表現式に対してcompile が正しい変換をすることを証明している（証明は省略した）のだが、こうした記述は、プログラムについての様々な主張を形式的に証明することができる。

Syntax

二項演算 binop は、PlusとTimes からなる

```
Inductive binop : Set := Plus | Times.
```

式 expは、定数式Constと二項演算式Binopからなる

```
Inductive exp : Set :=
| Const : nat -> exp
| Binop : binop -> exp -> exp -> exp.
```

Semantics

```
Definition binopDenote (b : binop) :
      nat -> nat -> nat :=
  match b with
    | Plus => plus
    | Times => mult
  end.
```

二項演算 binopの表示的意味
ふたつの自然数をとって
和または積の自然数を返す

```
Fixpoint expDenote (e : exp) : nat :=
  match e with
    | Const n => n
    | Binop b e1 e2 =>
       (binopDenote b) (expDenote e1) (expDenote e2)
  end.
```

式 expの表示的意味
Constの場合とBinopの場合

実行サンプル

Eval simpl in expDenote (Const 42).

(** [= 42 : nat] *)

24 => 24

Eval simpl in expDenote (Binop Plus (Const 2) (Const 2)).

(** [= 4 : nat] *)

(+ 2 2) => 4

Eval simpl in expDenote (Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7)).

(** [= 28 : nat] *)

(* (+ 2 2) 7) => 28

Target Language Syntax

Stack Machine の命令 instrの シンタクスの定義

```
Inductive instr : Set :=
| iConst : nat -> instr
| iBinop : binop -> instr.

Definition prog := list instr.
Definition stack := list nat.
```

listで二つのスタック progとstackを実装する

- progは、このStack Machineのプログラムを格納する
- stackは、計算作業用のスタック

Target Language Semantics

```
Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
    | iConst n => Some (n :: s)
    | iBinop b =>
      match s with
        | arg1 :: arg2 :: s' =>
            Some ((binopDenote b) arg1 arg2 :: s')
        | _ => None
      end
  end.
```

instrの表示的意味

定数なら作業用スタックに積む

iBinopならスタックの
先頭の二項を演算して、
結果をスタックに戻す。

Target Language Semantics

```
Fixpoint progDenote (p : prog) (s : stack) : option stack :=
  match p with
    | nil => Some s
    | i :: p' =>
      match instrDenote i s with
        | None => None
        | Some s' => progDenote p' s'
      end
  end.
```

progの表示的意味

Translation

式expをprogに変換する

```
Fixpoint compile (e : exp) : prog :=
  match e with
    | Const n => iConst n :: nil
    | Binop b e1 e2 => compile e2 ++ compile e1 ++
iBinop b :: nil
  end.
```

Translation
実行サンプル

Eval simpl in compile (Const 42).

(** [= iConst 42 :: nil : prog] *)

42 => (42) :prog

Eval simpl in compile (Binop Plus (Const 2) (Const 2)).

(** [= iConst 2 :: iConst 2 :: iBinop Plus :: nil : prog] *)

(+ 2 2) => (2 2 +) :prog

Eval simpl in compile (Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7)).

(** [= iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil : prog] *)

(* (+ 2 2) 7) => (7 2 2 + *) :prog

Translation Correctness

```
Theorem compile_correct : forall e,
  progDenote (compile e) nil =
      Some (expDenote e :: nil).

Lemma compile_correct' : forall e p s,
  progDenote (compile e ++ p) s =
    progDenote p (expDenote e :: s).
```

Coqでの、この証明の詳細は、省略する。

拡張

```
Inductive type : Set := Nat | Bool.

Inductive tbinop : type -> type -> type -> Set :=
| TPlus : tbinop Nat Nat Nat
| TTimes : tbinop Nat Nat Nat
| TEq : forall t, tbinop t t Bool
| TLt : tbinop Nat Nat Bool.
```

先のマシンに、演算として 等号と不等号を（いずれもBool値を返す）を加えて拡張する。

```
Inductive texp : type -> Set :=
| TNConst : nat -> texp Nat
| TBConst : bool -> texp Bool
| TBinop : forall t1 t2 t, tbinop t1 t2 t -> texp t1 ->
texp t2 -> texp t.

Definition typeDenote (t : type) : Set :=
  match t with
    | Nat => nat
    | Bool => bool
  end.
```

```
Definition tbinopDenote arg1 arg2 res (b : tbinop
arg1 arg2 res)
  : typeDenote arg1 -> typeDenote arg2 ->
typeDenote res :=
  match b with
    | TPlus => plus
    | TTimes => mult
    | TEq Nat => beq_nat
    | TEq Bool => eqb
    | TLt => leb
  end.
```

```
Fixpoint texpDenote t (e : texp t) : typeDenote t :=
  match e with
    | TNConst n => n
    | TBConst b => b
    | TBinop _ _ _ b e1 e2 => (tbinopDenote b)
(texpDenote e1) (texpDenote e2)
  end.
```

実行例

```
Eval simpl in texpDenote (TBinop TTimes (TBinop
TPlus (TNConst 2) (TNConst 2))
  (TNConst 7)).
(** [= 28 : typeDenote Nat] *)

Eval simpl in texpDenote (TBinop (TEq Nat) (TBinop
TPlus (TNConst 2) (TNConst 2))
  (TNConst 7)).
(** [= false : typeDenote Bool] *)

Eval simpl in texpDenote (TBinop TLt (TBinop TPlus
(TNConst 2) (TNConst 2))
  (TNConst 7)).
(** [= true : typeDenote Bool] *)
```

Deep Specificationでの仕様定義

Deep Specification での「仕様」について

- Deep Specificationを何か形式的な仕様から何か具体的なプログラムの実装を自動的に生成する技術だと考えていると、なかなかイメージが掴めなくなる。

- Deep Specificationの仕様は、動かない抽象的なものではなく、それ自身プログラムとして動作するものだ。Deep Specificationでは、仕様と実装の差は、そのプログラムが一般的か特殊的かの差であって、動くか動かないかに違いがあるわけでは無い。

Coqによる仕様の記述

- ある問題をコンピュータで解こうとする時、我々は様々なデータ構造やアルゴリズムを使ってプログラムを書く。Deep Specificationの「仕様」もこれと全く同じように書かれる。それは「実装」と言ってもいい。

- ただ、「実装」する言語が違うのだ。その言語は、プログラム自身についての形式的推論を可能にする能力を持つもの、例えば、Coqを用いる。こうした能力は、プログラムの検証で大きな役割を果たす。

- 様々なデータ構造やアルゴリズムを使って、ある問題を解くプログラムをCoqで「実装」してみる。それが、Deep Specificationの「仕様」の第一歩だと考えていい。

- ただ、プログラムで「実装」された「仕様」（それは、文字で書かれた仕様書より「形式的」なものと考えられる）ができれば、それで終わりなのでは無い。

- この「仕様＝実装」から、実行用の高速なコード（OcamlやCやVerlogのコード）が生成されることになる。こちらを、形式的「仕様」から導出された「実装」と考えることもできる。

- 次に、Deep Specificationのもっとも成功したプロジェクトの一つであるチッパラのKamiプロジェクトを例に、仕様と実装の関係を、別の角度から紹介したい。

Kami：仕様定義サンプル

https://github.com/sifive/Kami/tutorial.v

Kami Project

- チッパラのKamiプロジェクトは、RISC/Vチップの Coqで書かれた「仕様」からRISC/V「チップ」を生成する。（正確には、RISC/Vチップのハード記述言語Bluespecのコードを吐き出すのだが）

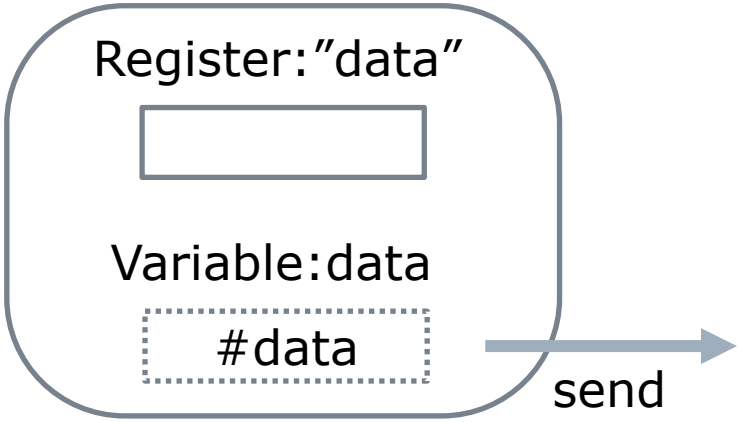
- ここでは、https://github.com/sifive/Kami/tutorial.v の内容を紹介する。

producer と consumer
二つのモジュールの実装

- まず、二つのモジュール producerと consumer を実装する。producerモジュールは、レジスター "data" を持ち、その値を変数data に読み込んで、その値を外部に送信する。送信後レジスターの値は更新され、同じ動作を繰り返す。consumerモジュールは受け取った値を外部に送信する。

Producer モジュールの実装

Module: producer



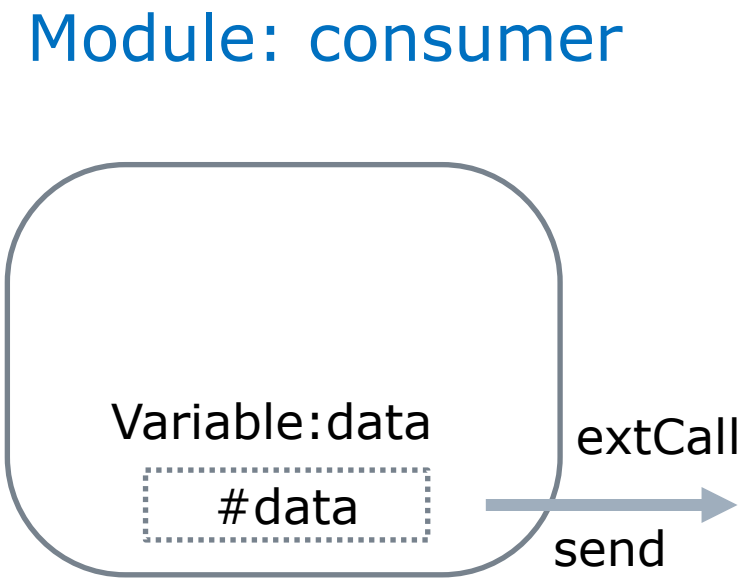
send後
Write "data" <- #data + $1

Producer モジュールの実装

```
Definition producer :=
  MODULE {
    Register "data" : Bit 32 (* type *) <- Default (* initial value *)

    with Rule "produce" :=
      Read data <- "data";
        (* Explicit actions are required to read values of registers into
local variables. *)
      Call (MethodSig "send" (Bit 32 (* parameter type *)): Void (*
return type*)) (#data) (* using [#] to read from variables *);
        (* Note embedding of a type assumption for the function
being called, not just its name. *)
      Write "data" <- #data + $1 (* using [$] for a literal expression
*);
      Retv
  }.
```

consumer モジュールの実装

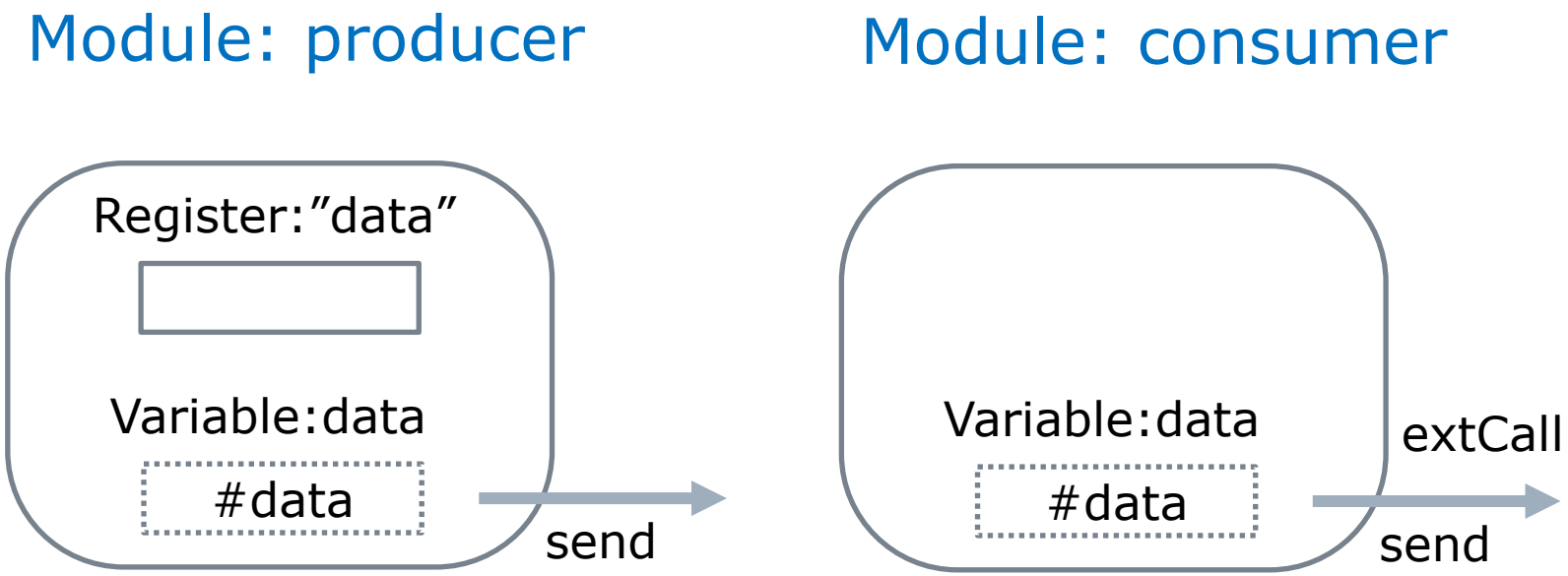


consumer モジュールの実装

```
(** For proof automation, it is recommended to register module definitions to "ModuleDefs". *)
Hint Unfold producer : ModuleDefs.

(** Consumer only has one method, which takes the data sent by Producer and calls an external function with the data. *)
Definition consumer :=
  MODULE {
    Method "send" (data: Bit 32): Void :=
      Call (MethodSig "extCall" (Bit 32): Void) (#data);
      Retv
  }.
```

producerモジュールとconsumer モジュール



send後
Write "data" <- #data + $1

二つのモジュールの結合

- Kamiでは、二つの実装モジュールを結合して一つの実装モジュールを構成できる。この実装モジュールをproducerconsumerImpl と呼ぼう。

- この操作をKamiでは、次のように表現できる！

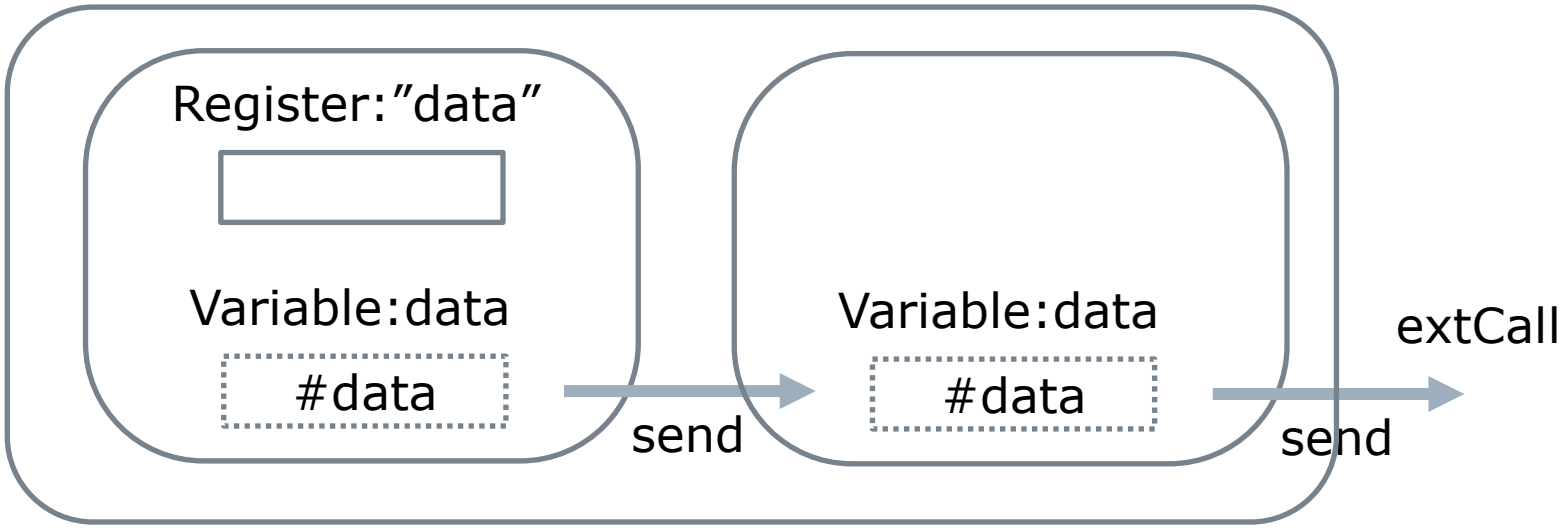
```
Definition producerconsumerImpl :=
  (producer ++ consumer)%kami.
```

- これはBluespec等のハードウェア記述言語の「回路合成」の考え方と同じものだと思う。

producerモジュールとconsumer モジュールの実装を組み合わせてproducerConsumerImplを作る

Definition producerConsumerImpl := (producer ++ consumer)%kami.

Module: producerconsumerImpl



send後
Write "data" <- #data + $1

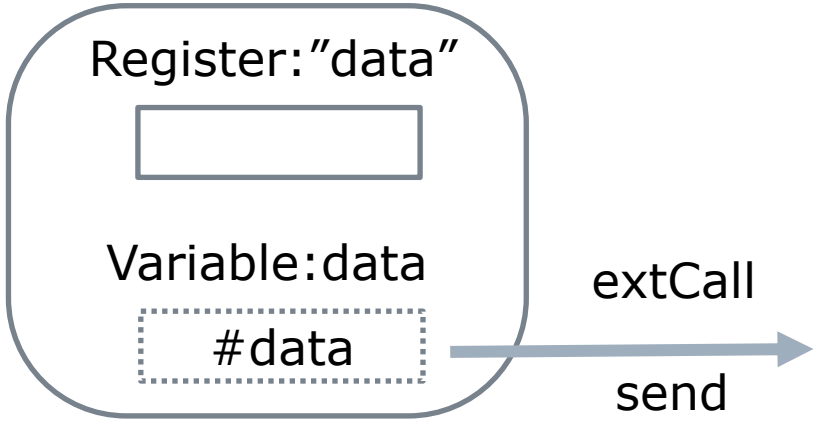
producerconsumerSpec

- それとは独立に、定期的に外部にデータを送信するモジュールを実装して、それを producerconsumerSpec と名付ける。このモジュールに、"Spec" 「仕様」という名前がついているのに注意してほしい。

- この時、先の「実装」 producerconsumerImpl が、この「仕様」producerconsumerSpec　の実装であることが、Coq上で「証明」できるのだ！

producerConsumerImplの仕様
producerConsumerSpecを書く

Module: producerconsumerSpec



send後
Write "data" <- #data + $1

producerConsumerImplの仕様
producerConsumerSpecを書く

```
Definition producerConsumerSpec :=
  MODULE {
    Register "data" : Bit 32 <- Default

    with Rule "produce_consume" :=
      Read data <- "data";
      Call (MethodSig "extCall" (Bit 32): Void) (#data);
      Write "data" <- #data + $1;
      Retv
  }.
```

実装producerConsumerImplは、
仕様producerConsumerSpecを
実装したものかをチェックする

次の定理を証明する。

```
Theorem producer_consumer_refinement:
  producerConsumerImpl <<== producerConsumerSpec.
```

Proof.
　(次のページ)
Qed.

きちんと証明できる。

Theorem producer_consumer_refinement:

```
Theorem producer_consumer_refinement:
  producerConsumerImpl <<== producerConsumerSpec.
Proof.
  kinline_left implInlined.
  (* Inlining: replace internal function calls in [impl]. *)
  kdecompose_nodefs producer_consumer_regMap
producer_consumer_ruleMap.
  (* Decomposition: consider all steps [impl] could take, requiring
that each be matched appropriately in [spec]. *)
  kinvert.
  (* Inversion on the took-a-step hypothesis, to produce one new
subgoal per [impl] rule, etc. *)
  kinv_magic_light.
  (* We have only one case for this example (for the one rule), and
it's easy. *)
Qed.
```