解説

基般

定理証明支援系に基づく形式検証

―近年の実例の紹介とCoq入門―

アフェルト レナルド　産業技術総合研究所

▶ 定理証明支援系とは？

実装の誤りによって，ソフトウェアの脆弱性が生じる．同様に，ディジタル社会に欠かせないセキュリティプロトコルなどは，高度な数学に基づくので，その数学に誤りが発見されれば，情報漏洩などの原因になる恐れがある．このようなことから，人間が書いたプログラムと数学の証明と両方の「正しさ」を保証してくれるシステムの開発は，重要な研究課題となった．定理証明支援系は，まさにそのようなシステムの１つである．

定理証明支援系は正しい証明を記述するための補佐ツールで，このツールを使って与えられた言明に対する証明を書けば，証明が正しいことが保証される．検証ツールの中で，定理証明支援系の適用範囲は，最も広く信頼性も高い．たとえば，モデル検査という検証ツールの適用対象は有限システムに限るが，定理証明支援系は数学的帰納法を利用することで，無限システムの検証もできる．定理証明支援系の中核となる部分（カーネルという）は小さいため，検証ツール自身にバグがある可能性は低く，検証結果の信頼性は高い．そのカーネルは形式論理の長い研究を経て選定された公理であるので，理論的に誤った結論を導かないことが紙上で十分確認できる．しかしながら，その高い汎用性と信頼性の一方で，課題もある．定理証明支援系による証明の作成の際，すべての証明手続を明確に形式化しなければならないので，一般的に自動化は困難なのである．しかし，定理証明支援系の研究開発は1970年代から継続的に行われた結果，2000年代からクリティカルな基盤ソフトウェアの検証や膨大な数学の証明の形式化が可能となった．現在，定理証明支援系は産業界でも重要な役割を果たすようになり，学術界でもその有用性は広く認められるようになった．

定理証明支援系の研究は主に欧州で行われている．伝統的な国際学会ITP（Interactive Theorem Proving☆1）の最近の論文の内容を調べると，最も発表件数の多い定理証明支援系はCoq☆2である．Coq[1]は，1984年からフランス国立情報学自動制御研究所（INRIA）で開発が進められ，多くの基礎的な研究の成果と具体的な応用（後で説明する数学の定理やコンパイラなど）が発表されている．また，その成果は，形式検証の世界だけでなく，プログラミング言語の世界でも利用されている．具体例として，ACM（Association for Computing Machinery）の有名な国際会議POPL（Principles of Programming Languages）の最近の論文の内容を調べると，2012年と2013年の20%以上の論文は定理証明支援系を利用し，その中でCoqは最も多く使われていた．Coqは，プログラミング言語とシステムの研究への貢献を讃えられ，2013年にACMのProgramming Languages Software賞を受賞した．Coq以外の定理証明支援系に，Isabelle（主にドイツとイギリスで開発），HOL（イギリス），ACL2（アメリカ）などもある．これらはCoqと異なる特徴を持ち，それぞれが重要な成果を導いているため，形式検証を行う際は，目的に適した定理証明支援系の選択が必要である．

☆1　旧名：Theorem Proving in Higher-Order Logics
☆2　http://coq.inria.fr/

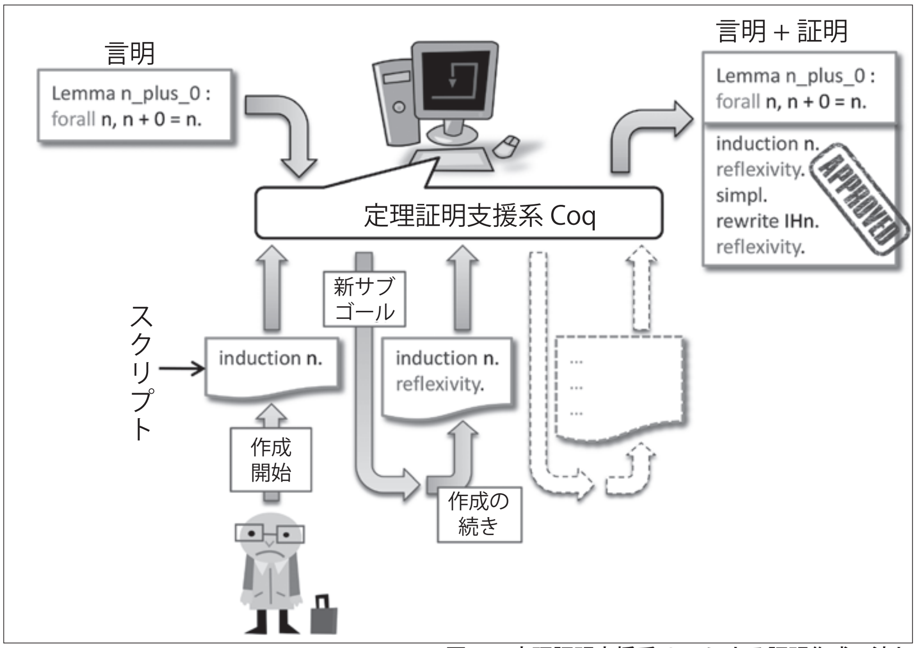

図 -1　定理証明支援系 Coq による証明作成の流れ

定理証明支援系では，対話的に証明を構成する．**図 -1** に，定理証明支援系 Coq を想定した証明の対話的な作成を示す．まず，言明の入力によって，証明のゴールが定まる．次に，ユーザが証明のスクリプトの記述を始める．その証明の最初のステップを Coq に入力すると，元々のゴールに辿り着くための新たなゴールが返される．それぞれのゴールに対して，ユーザがスクリプトの記述を続ける．このプロセスを繰り返すと，段階的に証明の作成が続き，最終的に Coq がゴールを返せなくなる．その時点で証明が完了し，Coq による保証が付く．

本稿の後半で，具体例を用いて，定理証明支援系 Coq による証明の作成を具体的に説明する．その前に，定理証明支援系のいくつかの代表的な実例を紹介する．

▶ 定理証明支援系による実例

定理証明支援系の応用は大きく分けて数学とプログラミング言語にある．歴史的に考えると，数学への応用は当然である．そもそも定理証明支援系の研究は型理論と関係が深い．型理論は 1900 年代に集合論で見つけられたパラドックスを避けるため，Russell と Whitehead が 1910 年代に導入した理論である．現在の型理論とは異なる体系であったが，動機は違わない．近年まで型理論は数学界にあまり影響を与えなかったが，定理証明支援系の最近の進歩でそれが変わってきた．またプログラミング言語の研究と定理証明支援系の基礎と開発は密接な関係にあり，プログラミングへの応用も自然な流れである．ほとんどの定理証明支援系のカーネルは，現代的なプログラミング言語（Haskell や Java など）のような型付き言語に基づく．代表的な型付きプログラミング言語 ML☆3 は，そもそも 1970 年代の Milner の定理証明支援系 LCF（Logic for Computable Functions）の開発から生まれた．定理証明支援系と型付き言語との関係は，今回の解説の後半（Coq の入門）で説明する．まず，実例の紹介から始めよう．

数学の証明の形式化

数学界への定理証明支援系の貢献は多い．特に，2005 年の四色定理の形式化は話題になった．四色定理というのは，いかなる地図も隣接する領域が異なる色になるように塗るには 4 色あれば十分だという定理である．1852 年にイギリスで Guthrie により言明されたが，De Morgan や Lebesgue などの有名な数学者の試みにもかかわらず，20 世紀後半まで正しく証明されなかった．1976 年にイリノイ大学の Appel と Haken が四色定理を証明したが，証明の一部は計算に約 2 カ月間かかる IBM 370 のアセンブリのコンピュータプログラムに任されていたため，一部の数学者から批判があった．加えて，その証明は大量の場合分けに基づくため，その正しさは簡単に確認できず，疑われていた．実際に，間もなく，場合分けとアセンブリに誤りが発見された．1995 年に Appel と Haken の証明が簡素化され，コ

☆3　ML は Meta-Language の略である．現在広く使われている Standard ML と OCaml が ML の実装である．

ンピュータプログラムも改善された（C 言語によるプログラムの実行は，当時の PC で約 3 時間）．しかし，コンピュータプログラムが本当に期待通りに動いていたという裏付けはまだなかった．2000 年以降 Gonthier と Werner は INRIA とマイクロソフトリサーチで定理証明支援系 Coq を用いて四色定理の形式化に取り組み始めた．その結果, 2004 年に, 四色定理の紙上の証明だけでなく，コンピュータの計算の正しさも数時間で検証可能になったのである．元々, 四色定理の形式的な言明は, 次のように短い:

```
Theorem four_color : forall m,
simple_map m -> map_colorable 4 m.
```

ここで，`simple_map` と `map_colorable` は 30 行以内のスクリプトで形式定義が書けるが，四色定理を証明するために，60,000 行のスクリプトが必要であった☆4．ともかく，2004 年の Gonthier らの形式化によって，ようやく信頼性の高い四色定理の証明を得ることができた[2)]．

四色定理の証明は難しく，歴史的には重要であるが，パズルのような問題であったため，その次の問題として，2005 年に Gonthier らは奇数位数定理の形式化に取り組んだ．奇数位数定理とは奇数位数の群が可解群だという定理である．1911 年に Burnside が予想し，1963 年に Feit と Thompson が証明した．この時代の群論の結果としては証明がかなり長かったため，数学者が証明の簡素化に努めたが，1990 年代になっても，その証明はまだ 250 ページと長かった．奇数位数定理の証明には，大学で教えられる群論や線形代数学などのほかに，大学院レベルのさまざまな理論も必要となるため，形式化は難しい．しかし，この証明に成功すれば，ほかの問題の解決に再利用可能な形式化のモジュールと技術が得られるため，多数の数学者が興味を持った．今回，多くの協力者の支援を得ることで（この分野では珍しく，学会論文の共著者は 15 人），7 年の研究を経て，2012 年 9 月に奇数位数定理の形式化が無事に完成した．結果の形式化は，150,000 行の Coq のスクリプトとなる．奇数位数定理の証明自体（その他は基礎の形式化となる）は 40,000 行であり，紙上の証明に比べて 4.5 倍しか大きくなっていない[3)]．その比率は期待通りである（伝統的に，形式化して大きくなる 4 倍程度の比率は De Bruijn 係数と呼ぶ．De Bruijn は 1968 年から Automath という最初の定理証明支援系を構築した）．以上の研究が評価され，Gonthier は 2011 年 EADS Foundation 賞を受賞した．

Gonthier らが奇数位数定理のために開発したライブラリを用いて，筆者は，情報理論の基礎となる Shannon 定理の形式化[4)] を行い，そのライブラリの汎用性を確認できた．

定理証明支援系での形式化は，後片付けの役割ではない．現在の数学の証明が膨大になっているため，定理証明支援系は，さらに重要な役割を果たすようになってきた．Kepler 予想は代表的な例である．17 世紀に Kepler は無限の空間において同半径の球を敷き詰めたとき，最密な充填方法は面心立法格子であると予想した．1998 年に Hales が Kepler 予想の証明を発表し，Annals of Mathematics に投稿した．2005 年に論文になったが，数年の査読を経ても，査読者は証明の正しさを保証できていない[2)]．その証明は，300 ページと長く，40,000 行のコンピュータプログラムにも依存していた．したがって，Hales らは Flyspeck☆5 というプロジェクトを立ち上げて，定理証明支援系 HOL Light を用いて現在 Kepler 予想の証明の形式化に取り組んでいるが，20 年ぐらいかかると予測されている．

Hales のほかにも，定理証明支援系を用いて研究をしている有名な数学者は存在する．プリンストン高等研究所では，2002 年のフィールズ賞の Voevodsky を含む数学者のチームはホモトピー理論（連続的な変形の理論）に定理証明支援系を応用している．数学界への貢献として，Coq を用いて，ホモトピー理論の証明を De Bruijn 係数より短く書けることを示した（紙上の証明とその形式化

☆4 「定理証明支援系 Coq 入門」章で Coq の形式言語とスクリプトを詳しく説明する．

☆5 FPK は Formal Proof of Kepler conjecture の略である，http://code.google.com/p/flyspeck/

は同じサイズになる場合もある）．逆に，定理証明支援系への貢献もある．ホモトピー理論を用いて，定理証明支援系の型理論に意味を与えることができ，2009 年に新たな公理（Univalence Axiom という）を安全に加えられることが分かった．すなわち，Voevodsky らは型理論に基づく定理証明支援系とトポロジの間に密接な関係（ホモトピー型理論という）を発見したのである☆6.

以上の実例に見られるように，数学界において定理証明支援系は欠かせないツールになりつつある．

ソフトウェアの形式検証

数学界以外に，定理証明支援系はソフトウェアの検証に対しても重要な役割を果たすようになった．その影響は IT 業界にまで及ぶ．近年 IT 製品において，安全性の根拠を示すことが求められている．その認証取得は，競争的な強みとなるが，そのコストの負担は大きくなる一方である．たとえば，コンピュータセキュリティのための国際規格としてコモンクライテリア（ISO/IEC 15408）は有名である．コモンクライテリアは，IT 製品に対して，セキュリティを認証するための評価基準を定める．コモンクライテリアにおいて最も厳密な評価レベルは EAL7 と言い，その評価を取得するため，定理証明支援系の使用は不可欠である．たとえば，欧州では 2000 年代からスマートカードの評価に定理証明支援系がしばしば使われている[5)].

とりわけ，コンピュータシステムの安全性を保証するため，基盤ソフトウェアの形式検証が注目されており，特に近年は，コンパイラの形式検証が研究されている．具体的には，コンパイルの前のプログラムとコンパイルによって得たアセンブリは同じ動作をするかどうか，定理証明支援系を用いて保証する．たとえば，2004 年から INRIA で C コンパイラの形式検証が続いている．そのコンパイラ（Compcert という）は従来のコンパイラよりもバグが少ないことが示された[6)]．ただし，Compcert のバグはまだ検証されていないところで発見されたため，今後検証範囲を広げることによってさらなる信頼性の向上を期待できる．そして信頼性の高いコンパイラの応用先として組込みシステムがある．実際に，Compcert の最新の研究は Airbus 社などと共同で行われている．ちなみに，コンパイラの形式検証は長期間のプロジェクトとして考えるべきである．2013 年に Microsoft Research Verified Software Milestone 賞を受賞した際，Compcert の研究を主に行っている Leroy が，2006 年当時は，「Compcert の検証の完成度は 80% ぐらいだ」と思っていたそうだが，2013 年に「振りかえってみると，20% にすぎなかったと認めなければならない」と語った．定理証明支援系による検証に必要な時間を予測するのは一般的に難しい．

C コンパイラより検証コストが多く必要となった基盤ソフトウェアの形式検証プロジェクトは，オーストラリアの NICTA 研究所で行われたマイクロカーネル seL4 のプロジェクトである．seL4 は主に C 言語で書かれた約 8,700 行のソースコードであり，定理証明支援系 Isabelle/HOL を用いて形式検証が行われた．形式仕様は抽象的なモデルであり，形式検証は Haskell のモデルを経て，C 言語の実装までの段階的な詳細化（refinement）によって行われた．このアプローチで対象となった約 7,500 行のソースコードの検証コストは，約 200,000 行のスクリプトに対して 25 人年であった（Compcert の場合，50,000 行のスクリプトに対して 4 人年だと思われる）．得られたマイクロカーネルは組込み用のオペレーティングシステムとしてビジネスと繋がる．seL4 の形式検証は計画的に行われたので，再現性に関して重要な情報を得た．具体的には，定理証明支援系を用いてコモンクライテリアによる評価を取得するためのコストの大ざっぱな見積もりができるようになった．EAL7 の評価を取得するため，1,000 行のソースコードは 100 万ドル以上かかると言われている．seL4 の形式検証の場合，1 行は 700 ドルかかった[7)] と言えるので，定理証明支援系による形式検証は，信頼性の最も高いソフトウェアの構築方法として，より経済的な方法になる可能性がある（1,000 行で 70 万ドル）．

seL4 より小規模であるが，定理証明支援系 Coq

☆6 http://homotopytypetheory.org/coq/

を用いて筆者も現実的なプログラムの形式検証を研究している．具体的に，セキュリティプロトコルの形式検証を目指して，暗号スキームの実装に必要なアセンブリで実装された算術関数[8]やC言語で実装されたネットワークパケット処理などの形式検証のために，検証基盤を開発している．

▶定理証明支援系 Coq 入門

前章の実例では，定理証明支援系 Coq が四色定理と奇数位数定理と Compcert の証明に使われた結果についてのみ紹介した．本章では，定理証明支援系 Coq 入門として，形式検証の具体例を紹介する．ただし，数学の形式化やコンパイラの正しさには定理証明支援系以外の専門的な技術が必要であるので，Coq の基礎のみ紹介する．

Curry-Howard 同型対応

定理証明支援系 Coq は，これから説明する Curry-Howard 同型対応に基づく．形式論理の最も基本的な論証は modus ponens だとよく知られている:「『A ならば B』が成り立ち A が成り立つ, ならば, B も成り立つ」．形式的に，modus ponens は次のように記述する：

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

一方，プログラミング言語の関数適用を考える．型 A の引数を与えたとき，型 B の値を出力する関数 f があるとする．当然，f に型 A を持つ a の値を渡すと，型 B の値を返す．つまり，「f が A → B という型を持ち a が A という型を持つ，ならば，f を a に適用した f a が B という型を持つ」．通常，値 a が型 A を持つことを「a : A」と記述し，関数適用に当たる型規則は次のように記述する：

$$\frac{f : A \to B \quad a : A}{f a : B}$$

明らかに，「：」の左側を無視すると，modus ponens が読める．実際にほかの論証も同様である．つまり，型付きプログラミング言語を用いて形式論理を表現できる．型付きプログラミング言語と形式論理の関係は 1930 年代に Curry が発見した．型付きプログラミング言語と形式論理の対応は偶然ではないことは 1969 年に Howard が明確にした．Curry-Howard 同型対応によって，「：」の左側の値は「証明」として解釈する．つまり，「a : A」は「値 a が型 A を持つ」より「a が言明 A の証明である」と考えればよい．

Curry-Howard 同型対応から，Coq の中核となる部分は Gallina☆7 と呼ぶ型付きラムダ式の実装である．ユーザにとって Haskell や OCaml などの通常の関数型プログラミング言語に見える．ただし，通常の関数型プログラミング言語より，型の表現力が高い．特に，Gallina に依存型という型の種類がある．たとえば，「**forall a : A, B a**」という型記述ができる（「**a**」に依存がない場合は,「**A → B**」と書く）．そういう型を持つ関数は，帰り値の型が入力の値に依存する．依存型が使えるので，Gallina の表現力は高く，形式モデルをコンパクトに表現できる．数学的な仕様の記述にも特に障害はない．ただし，これはあくまで表現力の話で，定理証明支援系の構築に依存型は不可欠ではない．たとえば，Isabelle という一流の定理証明支援系は依存型を扱わない．

定理証明支援系 Coq：システムの概要

前節では，証明が Gallina の式となり，言明は Gallina の型となることを説明した．通常，Gallina を用いて，基本的な定義（データ構造・関数・命題）を素直に記述できる．しかし，通常の関数型プログラミング言語 Haskell や OCaml などに比べて，Gallina の表現力が高いので，型推論が完全でなく，証明（すなわち，Gallina の式）の直接的な記述が困難である．証明の記述のために新たな道具が要り，その道具の組合せは Coq システムと呼ぶ（**図 -2** 参照）．まず，ユーザが証明として Gallina

☆7 ちなみに，Coq はフランス語の雄鶏という意味を持つ．Gallina は Gallinacé（キジ目）の省略だと思われる．

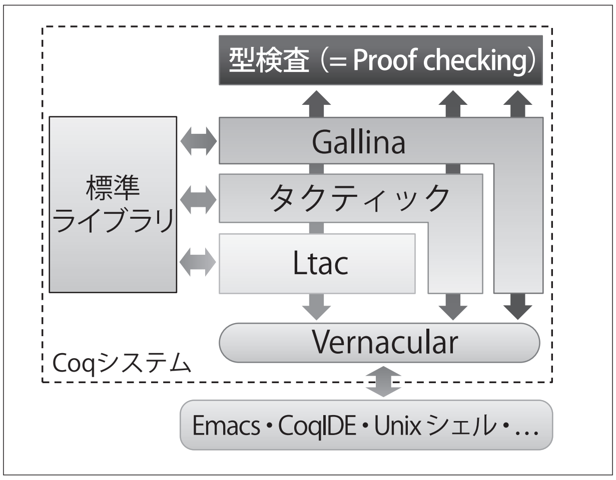

図 -2　定理証明支援系 Coq のシステムの概要

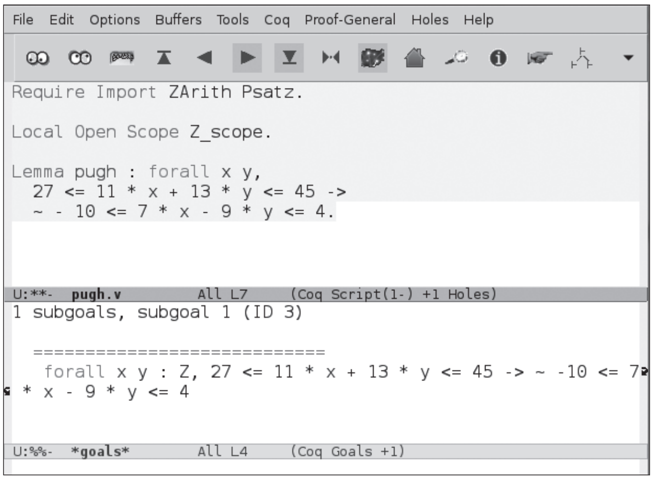

図 -3　証明の構成の例（1/3）

の式をほとんど書かず，代わりにタクティックを使う．タクティックは証明の半自動的な構成を行うものであり，形式論理のさまざまな基本的な論証手法を表現するものである．連続したタクティックはスクリプトと呼ぶ．完成したスクリプトは証明だと思ってもほとんど間違いはないが，正確に言うと証明の間接的な記述方法である．スクリプトの冗長度を減らすために，Ltac というプログラミング言語を用いて，タクティックの自動的な適用ができる．実際にタクティックは Ltac の要素として考えてもよい．直接に Gallina で，または間接にタクティックや Ltac で構築する証明を型検査しなければならない．型検査との対話は Vernacular という言語で行う．Gallina と Ltac と違って，Vernacular はプログラミング言語ではない．Vernacular の命令を用いて，定義や証明などを型検査に提出する．または，型検査のアルゴリズムの調整をする（たとえば，表記の設定，部分的な型推論の設定など）．これからいくつかの具体例を用いて，さまざまなタクティックや Vernacular の命令を紹介する．Coq システムはよく使われる定義と定理（たとえば，整数論，代数学, 実解析など）も標準ライブラリとして提供する．

定理証明支援系 Coq とユーザの間のやりとりは基本的にテキストで行われる．Unix シェルだけを通じて証明の構成ができるが，一般的に作業はカスタマイズ可能なテキストエディタで行われる．一番人気なインタフェースは emacs エディタの Proof General モードである．Coq を設定すると，CoqIDE という専用エディタも設定される．設定が面倒であれば，ProofWeb[☆8] は Coq の Web ブラウザインタフェースを提供するが，すべての標準モジュールが扱えるわけではない（たとえば，後で出てくるタクティック lia は現時点提供されていない）．

Coq での証明の構成

これから，整数に関する具体的な証明を記述してみる：

```
Lemma pugh : forall x y : Z,
  27 <= 11 * x + 13 * y <= 45 ->
    ~ - 10 <= 7 * x - 9 * y <= 4.
```

「Lemma 名前 : 言明.」は Vernacular の構文である．それを入力すると，Coq が言明の型検査を行い，証明モードに入り，ゴールとして言明を出力する（**図 -3** 下参照）．言明の中の「*」は整数の掛け算，「<=」は比較演算子，「~」は否定を表す．それぞれは Coq の標準ライブラリから得る．Coq の標準ライブラリで整数は Z という型を持ち，使用の際, Require Import ZArith の Vernacular の命令を実行する（図 -3 上参照）．整数に関する表記は Local Open Scope Z_scope の Vernacular の命令によって使えるようになる．

言明を入力した後，タクティックを用いて，証明を構成する．まず，タクティック intros x y H で仮定に x, y, H という名前を付け，ゴールの更新を行う（**図 -4** 参照）．

☆8　http://prover.cs.ru.nl

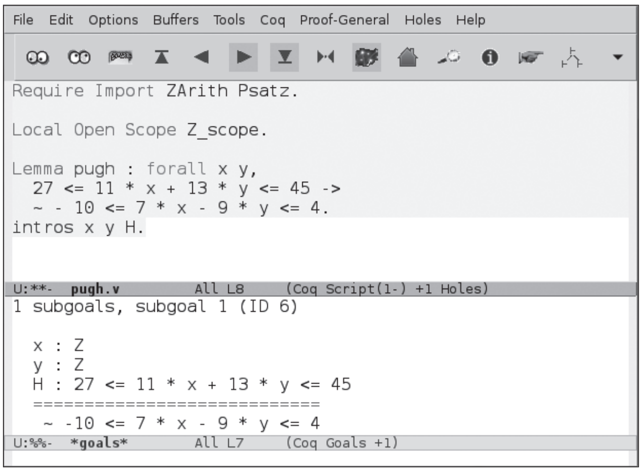

図 -4　証明の構成の例（2/3）

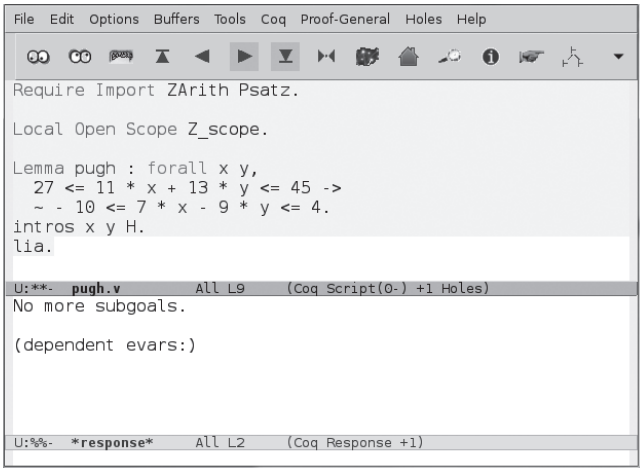

図 -5　証明の構成の例（3/3）

これから，仮定 `x` と `y` と `H` を用いて，`~ -10 <= 7 * x - 9 * y <= 4` を証明しなければならない……．幸いに，整数に関する定理の証明の自動生成のために，標準ライブラリの中にいくつかのタクティックが用意されている．たとえば，モジュール `Psatz` のタクティック `lia` で一次不等式を解くことができる（**図 -5** 参照）．

これで証明が完了である．スクリプトは「`intros x y H. lia.`」ですんだ．「`Qed.`」という Vernacular の命令を実行すると，証明に名前が付けられ，Coq に記録される．今回の定理の証明は上記のスクリプトだと思えばよい．正確に言うと，スクリプトは証明を構成するだけであって，証明自体を見るために，`Show Proof` という Vernacular の命令が使えるが，今回の証明はここで載せられないほど大きい．

証明の簡単な記述のために，定理証明支援系 Coq の標準ライブラリはいくつかの賢いタクティックを提供する：一次不等式（`lia`），等式（`congruence`），直観命題論理（`tauto`），一階述語論理（`firstorder`），Presburger 算術（`omega`），など．このタクティックの本来の目的は証明の対話的な構成を補助することである．

Coq での証明の対話的な構成

前節の証明の構成には賢いタクティックを使ったので，短いスクリプトですんだ．もちろん，一般的に，証明の構成は完全に自動になるのはまれである．以降，簡単な例を用いて，Coq による証明の対話的な構成の基礎を説明する．まず自然数の定義を説明する．

準備：自然数の足し算

Coq のデータ構造には `Set` という型がある．標準ライブラリの自然数 `nat` の型は Peano 整数のように定義する：

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

`O`（「オー」）というのは数字の「0」を表す．その他の自然数は関数型 `nat->nat` を持つ `S` で構築する：`S O` は「1」を表す，`S (S O)` は「2」，など．

自然数の足し算は再帰関数として定義する：

```
Fixpoint plus n m :=
match n with
  | O => m
  | S n' => S (plus n' m)
end.
```

つまり，実行によって `plus O m` は `m` となり，`plus (S n') m` は `S(plus n' m)` となる．以降，便宜上，`plus n m` は `n + m` と書く．`nat` と `plus` は Gallina の式であり，`Inductive` と `Fixpoint` は Vernacular の構文である．通常のラムダ計算のように Gallina の式の実行ができる．たとえば，「1+1」（`plus (S O) (S O)`）を実行すれば，「2」（`S (S`

0))となる．実行によって同じ結果を導くので，Coq にとって「1+1」と「2」は同じものである．

例：自然数の足し算の性質

上記の足し算の場合，一番目の引数に対して再帰を行うので，実行によって `0 + n = n` という性質は明確であるが，`n + 0 = n` は定理として証明しなければならない：

入力：Lemma n_plus_0 : forall n, n + 0 = n.
出力：
```
======================
forall n, n + 0 = n
```

最初に，帰納法のためのタクティックを用いて，`n` に対して帰納法を行う．したがって，Coq が基本ケースと帰納のケースを分けて，2 つの新たなゴールを出力する：

入力：induction n.
出力 1：
```
===============
0 + 0 = 0
```
出力 2：
```
n : nat
IHn : n + 0 = n
===============
S n + 0 = S n
```

出力 1 の `0 + 0` と `0` は足し算の実行によって同じなので，タクティック `reflexivity` で同値関係ゴールが消える（「同値関係と書き換え」項でその仕組みを説明する）：

入力：reflexivity.

出力 2 の場合，帰納法の仮定を使えるよう，相応しい形にゴールを書き換える．まず，タクティック `simpl` を用いて，`plus` の実行を求める：

入力：simpl.
出力：
```
n : nat
IHn : n + 0 = n
===============
S (n + 0) = S n
```

つぎ，帰納法の仮定を用いて，ゴールの中の式 `n + 0` を `n` に書き換える：

入力：rewrite IHn.
出力：
```
n : nat
IH : n + 0 = n
================
S n = S n
```

出力のゴールは `reflexivity` で解ける．以上，`n_plus_0` の言明の証明が完了した．

タクティックの実装

前節に出てきたタクティックは魔法のように帰納法や書き換えなどを行うように見えるかもしれない．タクティックの実装を理解しなくても Coq を十分使えるが，帰納法と書き換えの仕組みが分かると，Curry-Howard 同型対応がさらに明確になる．

帰納法の仕組み

帰納法のタクティック `induction` の仕組みの理解を深めるために，説明を補充する．

「準備：自然数の足し算」項で Coq のデータ構造は `Set` という型を持つと説明した．同様に，Coq の命題は `Prop` という型を持つ式となる．たとえば，`nat->Prop` という関数型は自然数を 1 つ引数にとる述語を表すため，たとえば，「この自然数は素数である」という命題はこの型を持つ．自然数に関して，数学的帰納法を型として記述すると，次のようになる☆9：

```
forall P : nat -> Prop, (* 述語 *)
```

☆9 ちなみに，その式自体は（高階）命題なので，型 `Prop` を持つ．つまり，型 `Prop` は impredicative である．型 `Set` との主な違いである．

```
P O -> (* 基本ケース *)
(forall n, P n -> P (S n)) -> (* 帰納 *)
forall n, P n (* 結論 *)
```

上記の帰納法は依存型を使う一例である．依存型は帰納仮定 forall n, P n -> P (S n) に使われる．関数の型であるが，その関数の結果の型は入力 n によって異なる．依存型のおかげで，帰納法は通常の数学と同じ記述となる．

上記の型はあくまでただの言明である．本当に成り立つかどうか現時点ではまだ分からない．実は，自然数を定義した際に，Coq が帰納法の定理を裏で証明している☆10．具体的には，Curry-Howard 同型対応に沿って，次の Gallina の関数を帰納法の証明として構築した：

```
Fixpoint nat_ind (P : nat -> Prop) (P0 : P
0) (IH : forall n, P n -> P (S n)) (n : nat) :=
  match n with
   | O => P0
   | S m => IH m (nat_ind P P0 IH m)
  end.
```

少々難しく見えるかもしれないが，まず nat_ind という関数は帰納法の型を持つ関数だということが確認できる（P0 は基本ケース，IH は「induction hypothesis」）．これだけで数学的帰納法が証明できたことになる．nat_ind という関数は nat->Prop であるような命題 P を受け取って，P 0（0 の場合 P が成り立つこと）の証明を受け取って，forall n, P n -> P (S n) の証明を受け取って，forall n, P n の証明を構築する．nat_ind を実行すると，n は O の場合，基本ケース P 0 の証明 P0 を返す．それ以外の場合，帰納のケースと再帰関数呼出（帰納法の仮定に対応）を利用する．

タクティック induction は nat_ind の適用とほとんど同じである．たとえば，前節の n_plus_0 の証明では，n に対する帰納法を行うように，induction の代わりに apply nat_indというタクティックを使うと，ゴールと nat_ind の述語 P に対して単一化が行われた結果，P は fun x => x = n + 0 となり，2 つのゴール（基本ケース・帰納のケース）が出力されることになる．

☆10 nested datatypes, mutually recursive types，などのデータ構造の場合は少し手間がかかる．

同値関係と書き換え

タクティック induction のように，reflexivity と rewrite はタクティック apply に基づいてできている．

同値関係は，Gallina から直接得るものでなく，Inductive を用いて，Gallina で最小反射関係として定義する☆11：

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
refl_equal : eq x x.
```

（Type は Set と Prop よりも一般的な型である）．eq x y は x = y と記述する．eq の定義によって，x = y は成り立たなくても，いつでも言明ができる．しかし，x = yを証明する方法は 1 つしかない：refl_equal x という式である．つまり，Gallina の実行によって x と y は同じにならなければならない．したがって，reflexivity というタクティックは refl_equal の提供（すなわち，apply refl_equal）として考えてもよい．

同値関係の定義から成り立つ性質のうち，最も重要なのは，x=y が成り立つと，x を y に書き換えることができることである．そもそも同値関係は Inductive で定義されたので，nat のように帰納法も自動生成される：

```
eq_ind :
forall (A : Type) (x : A) (P : A -> Prop),
P x -> forall y : A, x = y -> P y
```

言い換えると，x=y の証明があれば，P x の証明

☆11 Coq の同値関係の定義によって，等しいものは同じ性質を持つ．そのような同値関係は Leibniz 等価性と呼ぶ．同値関係の形式化に関して疑問を持つ方がいるかもしれない．実際に，それぞれの定理証明支援系の実装がよく異なるところである．Coq の場合，2013 年現在同値関係の形式化と変換ルールとの関係はまだ研究対象となっている．

からP yの証明を作成できる．つまり，eq_indの適用は書き換えに相当する．たとえば，n_plus_0の証明の中にでてきたrewrite IHnはeq_indの適用として考えればよい．具体的に，その場合，rewrite IHn は apply (eq_ind n (fun x => S x = S n))とほとんど同じである．

さらに大きなスクリプトに向けて

証明の構成のために，対話的にスクリプトを記述しなければならない．数十万行のスクリプトの作成は珍しくはないため，膨大なスクリプトを管理するための技術が開発されている．四色定理と奇数位数定理のため，INRIA・マイクロソフトリサーチで数学の形式化向けのSSReflect☆12というCoqの拡張が開発されている．SSReflectではCoqの一番重要なタクティックの表現力が向上された．そのおかげで，スクリプトは，従来と比較すると短くなり，メンテナンスしやすくなった．このことは，形式化が大きくなればなるほど，重要な役割を果たす．

「Coqでの証明の構成」節で紹介した賢いタクティック（liaなど）は，対話的な証明の構成の負担を減らす方法の１つである．実際には，Coqのユーザは既存のタクティックの利用に限らず，新たなタクティックの開発もできる．「定理証明支援系Coq：システムの概要」節で紹介した言語Ltac（図-2参照）は１つのタクティック開発方法である．Ltacのプログラムの実行によって，既存のタクティックの自動的な適用をするので手続きの実装ができる．今回細かく説明しないが，reflectionという技術を使って，Gallinaでも証明の構成を自動化できるが，Ltacのほうが寛容な言語なので，Coqシステムでこの２つの言語があっても冗長ではない．

▶まとめ

今回，定理証明支援系による形式検証の主な実例について紹介した．定理証明支援系は，形式検証の世界を超え，コンピュータサイエンス全体と数学に影響を与え，その応用例は，産業界にまで広がっている．また，その技術的貢献の実例を具体的に把握できるように，代表的な定理証明支援系Coqについて，基礎的な内容を説明した．定理証明支援系による形式検証は，非常に高度な技術を要し，難しい作業に見えるかもしれないが，実は中毒性になるほど楽しく，流行しつつある．コンパイラの最適化は切りがなく，永遠に仕事があると言われているが，筆者は定理証明支援系による形式検証に関しても，同じように思う．たとえば，Compcertの形式検証は，その検証作業当初，速やかに終わると思われていたが，９年が経っても，新たな成果が次々と出，構築された検証基盤をほかの研究者が再利用し，さらに新しい研究課題を生み出している．また，想像されていなかった数学との関係も発見された．定理証明支援系は，確立された研究を見直すチャンスを与えるものであり，とても有望な研究課題である．

☆12 http://www.msr-inria.fr/projects/mathematical-components/

参考文献

1) The Coq Development Team, The Coq Proof Assistant Reference Manual, INRIA (1999-2013).
2) Hales, T. C., Gonthier, G., Harrison, J. and Wiedijk, F. : Formal Proof, *Notices of the AMS*, Vol.55, No.11, pp.1370-1414 (2008).
3) Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E. and Thery, L. : A Machine-Checked Proof of the Odd Order Theorem, in *Interactive Theorem Proving,* Rennes, France (2013).
4) Affeldt, R. and Hagiwara, M. : Formalization of Shannon's Theorems in SSReflect-Coq, in *Interactive Theorem Proving*, Princeton, NJ, USA (2012).
5) Chetali, B. and Nguyen, Q.-H. : About the World-first Smart Card Certificate with EAL7 Formal Assurances, in *9th International Common Criteria Conference*, Jeju, Korea (2008).
6) Yang, X., Chen, Y., Eide, E. and Regehr, J. : Finding and Understanding Bugs in C Compilers, in *Programming Language Design and Implementation*, San Jose, CA, USA (2011).
7) Klein, G. : From a Verified Kernel towards Verified Systems, in *Asian Symposium on Programming Languages and Systems*, Shanghai, China (2010).
8) Affeldt, R. : On Construction of a Library of Formally Verified Low-level Arithmetic Functions, *Innovations in Systems and Software Engineering*, Vol.9, No.2, pp.59-77 (2013).

（2013年8月27日受付）

AFFELDT Reynald（アフェルト レナルド）▌正会員
reynald.affeldt@aist.go.jp

2004年東京大学大学院情報理工科学研究科・コンピュータ科学専攻博士課程修了．現在，産業技術総合研究所・セキュアシステム研究部門主任研究員．定理証明支援系CoqによるアセンブリやC言語のプログラムの形式検証や情報理論の形式化などを研究．