定理証明支援系 Coq 入門

SSReflect 中心

アフェルト レナルド

産業技術総合研究所

2014 年 9 月 7 日 (日)

本発表

- Coq/SSReflect 入門
  - Coq (フランス国立情報学自動制御研究所)
  - SSReflect (フランス国立情報学自動制御研究所 + マイクロソフトリサーチ)
  - ➡ 簡単なインストール情報
- 発表者の経験 + Coq/SSReflect に関する学会発表等の抜粋より
  - 参考文献: スライド 113〜
- [Aff14] の内容を詳細化

Outline

定理証明支援系とは
定理証明支援系の概要
定理証明支援系の歴史
定理証明支援系の応用例
数学の証明の形式化
ソフトウェアの形式検証
定理証明支援系 Coq/SSReflect の入門
Coq による形式証明の原理
形式モデルと仕様の作成
Coq と SSReflect の関係
形式証明の基本
SSReflect ライブラリの紹介
SSReflect ライブラリの概要
基礎ライブラリ
総和と総乗
群と代数
関連するトピックス
命令型プログラムの検証
アセンブリ言語への応用
C 言語への応用
数学の証明の形式化
情報理論の形式化
符号理論の形式化

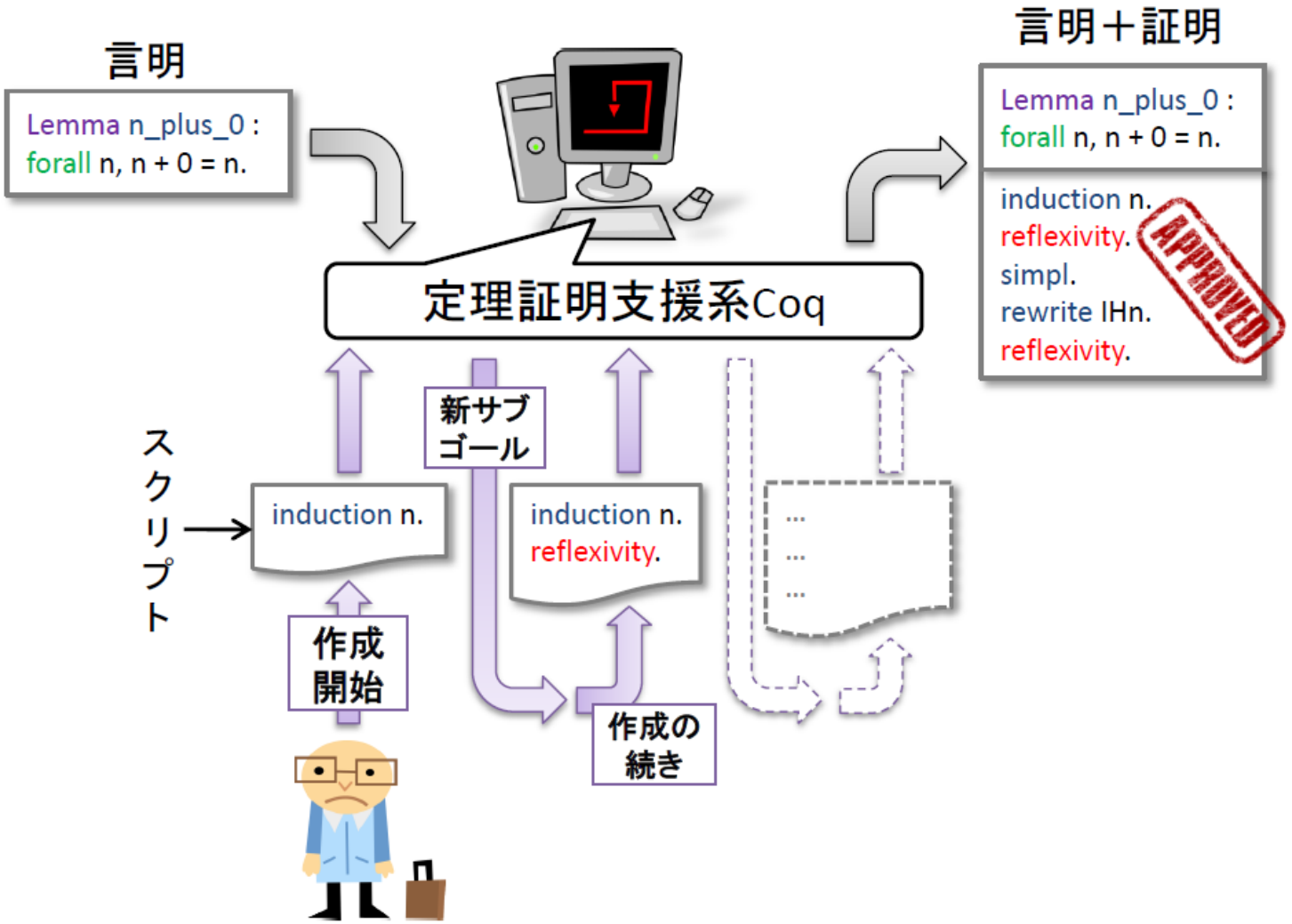
定理証明支援系による形式検証の動機

- ソフトウェア安全性の保証, バグがないことの保証
  - 問題の例: OpenSSL
    - Debian の弱い鍵 (2006 年〜2008 年)
    - Heartbleed (2014 年に発表)
  - ハードウェアへの応用 (J. Harrison)
- 数学の証明の正しさ
  - Kepler 予想の証明の査読 [Hal08] (スライド 13)
  - “A technical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail.” [Voe14]
- 安全な開発方法
  - 基盤ソフトウェア (例: CompCert コンパイラ [Ler09], seL4 マイクロカーネル [WKS+09])
  - 膨大な数学証明の時代:
    - Polymath プロジェクト
    - Kepler 予想の証明の形式化の国際協力 [Hal12]
    - “the future of both mathematics and programming lies in the fruitful combination of formal verification and the usual social processes that are already working in both scientific disciplines” [AGN09]

定理証明支援系とは?

- 定理証明支援系の役割: 証明の (1) 記述を支援, (2) 正しさを保証
  - 信頼性が高い:
    カーネル (中核部分) は小さいため, 理論的な誤りは紙上で確認できる
  - 汎用性が高い: 数学的帰納法を利用できるので, 有限システムに制限されない
- 型理論に基く定理証明支援系の例: Coq, HOL Light, Isabelle/HOL 等
  - その他の理論に基く定理証明支援系: Mizar (1973 年から, Tarski-Grothendieck 集合論に基く, 計算力ない), ACL2, PVS
- 対話的に証明を構成する:
  1. 言明の入力によって, 証明のゴールが定まる
  2. ユーザが証明のスクリプトの記述を始める
  3. 証明のステップを行うと, 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
  4. 段階的に証明の作成が続き, 最終的にゴールが返されなくなる

対話的な証明の流れ


言明
Lemma n_plus_0 :
forall n, n + 0 = n.
定理証明支援系Coq
言明＋証明
Lemma n_plus_0 :
forall n, n + 0 = n.
induction n.
reflexivity.
simpl.
rewrite IHn.
reflexivity.
APPROVED
スクリプト
induction n.
新サブゴール
induction n.
reflexivity.
...
...
...
作成開始
作成の続き

型理論に基づく定理証明支援系の歴史 I

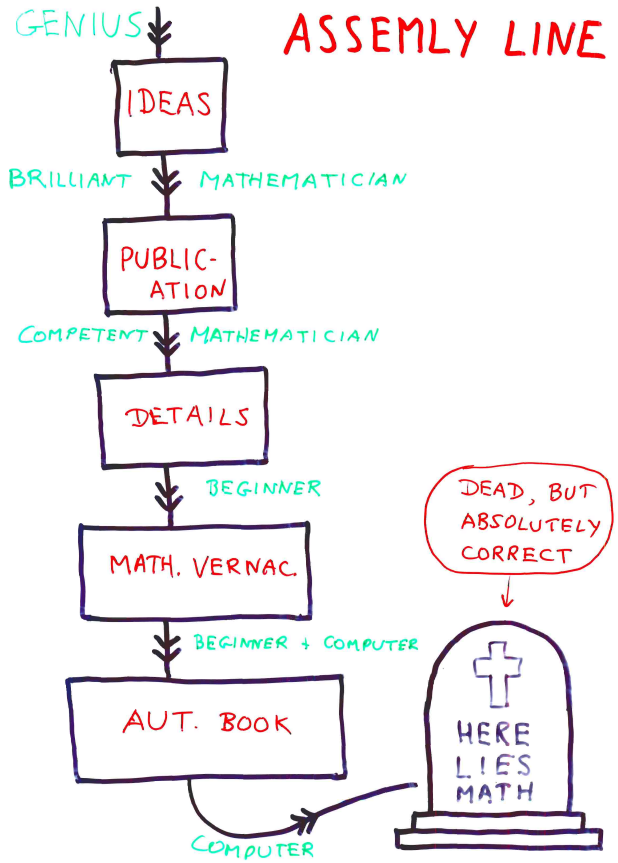
- 19 世紀: 数学の基礎の研究の開始 (1879 年: G. Frege の Begriffsschrift; 1880 年代: G. Cantor による集合論)
- 1901 年: B. Russell が集合論の簡単な矛盾を発見 ($a = \{x \mid x \notin x\}, a \in a \leftrightarrow a \notin a$) $\Rightarrow$ “It is the distinction between logical types that is the key to the whole mystery.” (以上については [vH02] に参照)
- 1908 年: B. Russell の “vicious-circle principle”: “Whatever contains an apparent variable must not be a possible value of that variable”[Rus08] $\Rightarrow$ 型の hierarchy (individuals < first-order propositions < $\cdots$)
- 1910–1913 年: B. Russell と A. N. Whitehead の Principia Mathematica; 型を用いた集合論による数学の再構築; 批判があった (L. Wittgenstein 等); 数学世界に影響はなかった
- 1930 年代: H. B. Curry が命題論理とコンビネータの間の Curry 同型対応を発見
- 1940 年: A. Church の Simple Theory of Types[Chu40]; 型付 λ 計算を利用 (型 ι: “individuals”; 型 o: “propositions”); extensional; 定理証明支援系 HOL の基礎

型理論に基づく定理証明支援系の歴史 II

- 1950 年代: カット除去と λ 計算の実効の間 (W. W. Tait)
- 1967–1968 年: N. G. de Bruijn, 定理証明支援系 AUTOMATH
- 1969 年: Curry-Howard 同型対応 [How80]: proof-checking = type-checking (スライド 31)
- 1972–1973 年: P. Martin-Löf の型理論; Leibniz equality を含む
- 1970 年代: R. Milner の LCF (Logic for Computable Functions); 型理論の機械化 ⇒ 型つきプログラミング言語 ML の発想
- 1984–1985 年: 定理証明支援系 Coq の開発の開始 [CH86]
- 2005 年から: クリティカルな基盤ソフトウェアの検証 (CompCert, seL4), 膨大な数学の証明の形式化 (四色定理, Kepler 予想)

数学の形式化

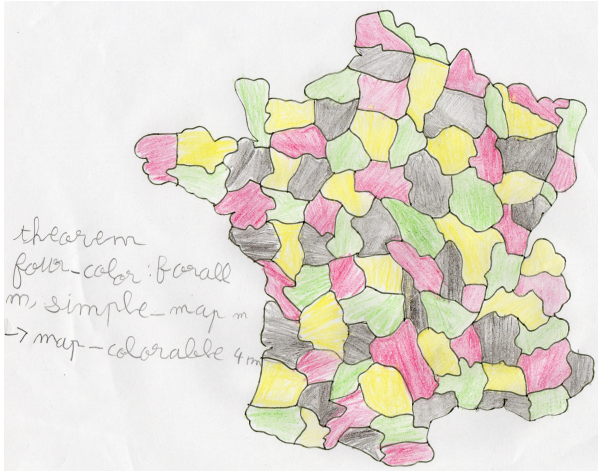
[dB03] による:



- 目的:
  - ミス, 穴のない証明の作成
  - 証明の最適化, 短さ
  - 関連する定理の発見に繋る
- 難しさ [Wie14]:
  - 教科書の 1 頁: 約 1 週間かかる
  - 200 頁の教科書: 約 5 人年かかる

四色定理の形式化 I

「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 色あれば十分である」



- 1852 年: F. Guthrie (イギリス) による言明
- 試み: A. de Morgan, H. L. Lebesgue 等
- 1976 年: K. Appel, W. Haken (イリノイ大学) による証明
  - 正しさは簡単に確認できないので, 一部の数学者から批判
    - 一部の計算は IBM 370 のアセンブリプログラムに任されていた (実行は約 2ヶ月間かかった), 大量の場合分け
    - 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そう
- 1995 年: 証明の簡単化; コンピュータプログラムの改善 (C 言語, 当時の PC で約 3 時間)

四色定理の形式化 II

- 2000 年ごろ: G. Gonthier と B. Werner (INRIA, Microsoft Research) は Coq で形式化を開始
- 2005 年: 四色定理の形式化が完成 [Gon08]
  - 数時間で検証可能
  - 言明: 30 行以内のスクリプトで形式定義:

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

  - 証明: 約 60,000 行のスクリプト [Gon05]
- SSReflect の開発のきっかけ

奇数位数定理

- 1911 年: W. Burnside による予想
- 1963 年: W. Feit と J. G. Thompson が証明
  - この時代の群論の結果として, 証明は長かった
  - 大学の群論や線形代数学等が必要, 大学院レベルの様々な理論も必要
- 1990 年代: 簡単化 ⇒ 証明: 255 ページ
- G. Gonthier らが Feit-Thompson 定理の形式化を取り組む
- (2011 年: G. Gonthier は EADS Foundation 賞を受賞)
- 2012 年 9 月: 完成

```
Theorem Feit_Thompson (gT : finGroupType) (G : {group gT}) :
  odd #|G| -> solvable G.
```

(`PFsection14.v`)

- 7 年間の研究, 多くの協力者 (学会論文 [GAA+13] は 15 人)
- 約 164,000 行の Coq のスクリプト (奇数位数定理の証明自体約 40,000 行); 紙上の証明に比べて 4.5 倍

Kepler 予想の証明の形式化 I

「無限の空間において同伴径の球を敷き詰めたとき, 最密な充填方法は面心立方格子である」

- 1611 年: J. Kepler が予想を発表
- 1998 年: T. C. Hales と S. P. Ferguson が証明を発表; Annals of Mathematics に投稿
- 証明: 300 ページ, 40,000 行のプログラム
- 2005 年: 論文発表; しかし, 4 年間の査読を経ても証明の正しさを保証出来ない [Hal08]
- 2003 年 (の数年後): Flyspeck (Formal Proof of Kepler Conjecture の略) プロジェクト開始

Kepler 予想の証明の形式化 II

- 形式化に必要な時間の見積は難しい:
  - 2008 年: “twenty work-years”? [Hal08]
  - 2011 年: 80% complete [Hal12]
  - 2014 年 8 月 10 日: 完成
    (`http://code.google.com/p/flyspeck/wiki/AnnouncingCompletion`)
- スクリプトのサイズ: 約 325,000 行と言われている
- 国際協力 (米国やベトナムやドイツ等からの開発者)
- その他の予想の証明 [Hal12]:
  - 1969 年の F. Tóth’s full contact 予想
  - 2000 年の K. Bezdek’s strong dodecahedral 予想
- 定理証明支援系
  - 一部は Isabelle/HOL: グラフの enumeration の問題
    - 2000 行の Java プログラムによるグラフ生成 (3 時間で, 23,000,000 個, 平均サイズ = 13 ノード)
    - 2011 年: 二つのグラフが足りてないことを発見 (Java プログラム最適化によるバグ; 健全性に影響なし) [Nip14]
  - 主に HOL Light
    - linear programming の結果の形式化

Kepler 予想の証明の形式化 III

- non-linear inequalities の検証 (multivariate polynomials, non-polynomials (arctan, sqrt, etc.); 元の C++プログラムは OCaml と HOL Light に移植) [Hal14]

- HOL Light 用の SSReflect: 5 つのタクティック
  - スライド 30 からタクティックの詳細な説明
  - 2 つのライブラリ → 短いスクリプト (2–3 times)

⇒ Flyspeck の 5-10% は SSReflect を使う [Hal14]

ホモトピー型理論と Univalent 基礎

- 近年, 定理証明支援系とトポロジの間の密接な関係が発見された
- ホモトピー型理論
  - ホモトピー理論 (連続的な変形の理論) を用いた型理論の解釈 (S. Awodey, M. A. Warren, 2005 年〜) [PW14]
  - 例えば,「$a : A$」$\stackrel{def}{=} a$ は A というスペースのポイント;「$p : a =_A b$」$\stackrel{def}{=} a$ と b の間のパス
  - 2014 年: “Carnegie Mellon Awarded $7.5 Million Department of Defense Grant To Reshape Mathematics”
- Univalent 基礎
  - プリンストン高等研究所の V. Voevodsky(2002 年フィールズ賞) によるプロジェクト
  - 型理論のモデルの開発の際, 2009 年に Univalence 公理の発見
    ⇒ 同型のものを等しいものとして見てもいい枠組
  - Univalence 公理で拡張した型理論で数学の開発
  - 型はスペースなので, 従来より低レベルではない
    ⇒ ホモトピー理論の証明は短く書ける (紙上の証明とその形式化は同じサイズになる場合がある)
  - Coq の UniMath ライブラリ (> 12,000 行)
    - ホモトピー理論の概念, abstract algebra の基礎の形式化 (“Foundations”)
    - 応用: 圏論の形式化 (B. Arhens, D. Grayson) 等

コモンクライテリアによる認証取得 I

定理証明支援系の影響は IT 業界まで及ぶ

- 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
- 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は有名
  - セキュリティを認証するための評価基準を定める
  - 1996 年から
  - ISO/IEC 15408
- 最も厳密な評価レベルは EAL7
  - その評価を取得するため, 定理証明支援系の利用は不可欠
- 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はしばしば使われている
  - 例: JavaCard (117,000 行の Coq スクリプト, 2003 年からの研究), Java CardAPI の複数のバグの発見 [CN08]
- EAL7 の評価取得例 (フランスの Trusted Labs 社と共同):
  - JavaCard の実装: SIMEOS (2007 年) と m-NFC (2012 年) (Gemalto, フランス)
  - Multos (2013 年)
  - Samsung のマイコンの MMU (2013 年)

コモンクライテリアによる認証取得 II

- 認証取得は, 競争的な強みとなるが, コストの負担は大きくなる
  - EAL7: 数千行のソースコードは数百万ドルかかると言われている [Bol10]
  - EAL6: 1 行, 1000 ドルとも言う [Kle10]

C コンパイラ (CompCert)

信頼性の高いコンパイラの構築

- コンパイルの前の C 言語のソースコード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
- 2004 年から INRIA で X. Leroy らが形式検証を続けている [Ler09, BL09]
- CompCert は従来のコンパイラよりバグが少ないことが示された
  - テストによる比較実験 [YCER11]
  - 発見された CompCert のバグはまだ検証されていないところにあった
- 応用先: 組み込みシステム (最新の研究は Airbus 社等と共に)
- 約 50,000 行のスクリプト, 4 人年だと言われている
- 最新の成果: Verasco's abstract interpreter (30,000 行)
- 2013 年: X. Leroy は Microsoft Research Verified Software Milestone 賞を受賞
- 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
  - 2013 年に受賞の際: X. Leroy:「2006 には (NB: 国際学会で CompCert の初発表), 検証の完成度は 80%ぐらいだと思っていた; 2013 年に振り返ってみると, 20%に過ぎなかったと認めなければならない」

seL4 マイクロカーネル

- オーストラリアの NICTA 研究所
- seL4 のソースコード: C 言語; 約 8,700 行
- 定理証明支援系: Isabelle/HOL
- アプローチ: (段階的な) 詳細化 (refinement)
  - 形式仕様は抽象的なモデル, 中間モデルは Haskell, C 言語の実装まで
- 7,500 行のソースコードに対し約 200,000 行のスクリプト, 約 25 人年 [Kle10]
  - 全部を含む: C 言語の検証基盤, tools, 定理のライブラリ等
- 組み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
- 2014 年 7 月からオープンソース
- seL4 プロジェクトは計画的に運営されたので, 重要な情報を得た:
  - 1 行は 700 ドル (1,000 行, 70 万ドル)
  - カーネルだけだと, 約 12 人年/1 行は 350 ドル
  - 再現性: 予想ではこれから似たプロジェクトの際, 12 人年/1 行は 230 ドル

⇒ 信頼性の最も高いソフトウェアの開発方法として, 形式検証はより経済的な開発方法になる可能性がある

Outline

定理証明支援系とは
- 定理証明支援系の概要
- 定理証明支援系の歴史
- 定理証明支援系の応用例
  - 数学の証明の形式化
  - ソフトウェアの形式検証

定理証明支援系 Coq/SSReflect の入門
- Coq による形式証明の原理
- 形式モデルと仕様の作成
- Coq と SSReflect の関係
- 形式証明の基本

SSReflect ライブラリの紹介
- SSReflect ライブラリの概要
- 基礎ライブラリ
- 総和と総乗
- 群と代数

関連するトピックス
- 命令型プログラムの検証
  - アセンブリ言語への応用
  - C 言語への応用
- 数学の証明の形式化
  - 情報理論の形式化
  - 符号理論の形式化

定理証明支援系 Coq

- 最も使われている定理証明支援系
  - 代表的な国際学会 ITP の論文の割合
  - プログラミング基礎の有名な国際学会 POPL(ACM) の論文の割合
    - 補佐ツールとして: 定理の正しさの確認, 検証フレームワークの開発基盤, プログラミング基礎の研究等
    - 2012 年と 2013 年に 20% 以上の論文は定理証明支援系を利用していた
- 開発の開始: 1984–1985 年
- 受賞:
  - ACM SIGPLAN Programming Languages Software 2013 賞
  - ACM Software System 2013 賞
- 基礎: Calculus of Inductive Constructions [CH88, PM92]
- カーネルのサイズ: 約 20,000 行 (約 2,500 行の C も含む OCaml) (NB: HOL Light: 約 400 行 [Har06], 帰納的に定義されている型 (ι-reduction 等) の違い)

Coq システムの原理

Proposition-as-types:

- 最も基本的な論証: Modus Ponens
  「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:

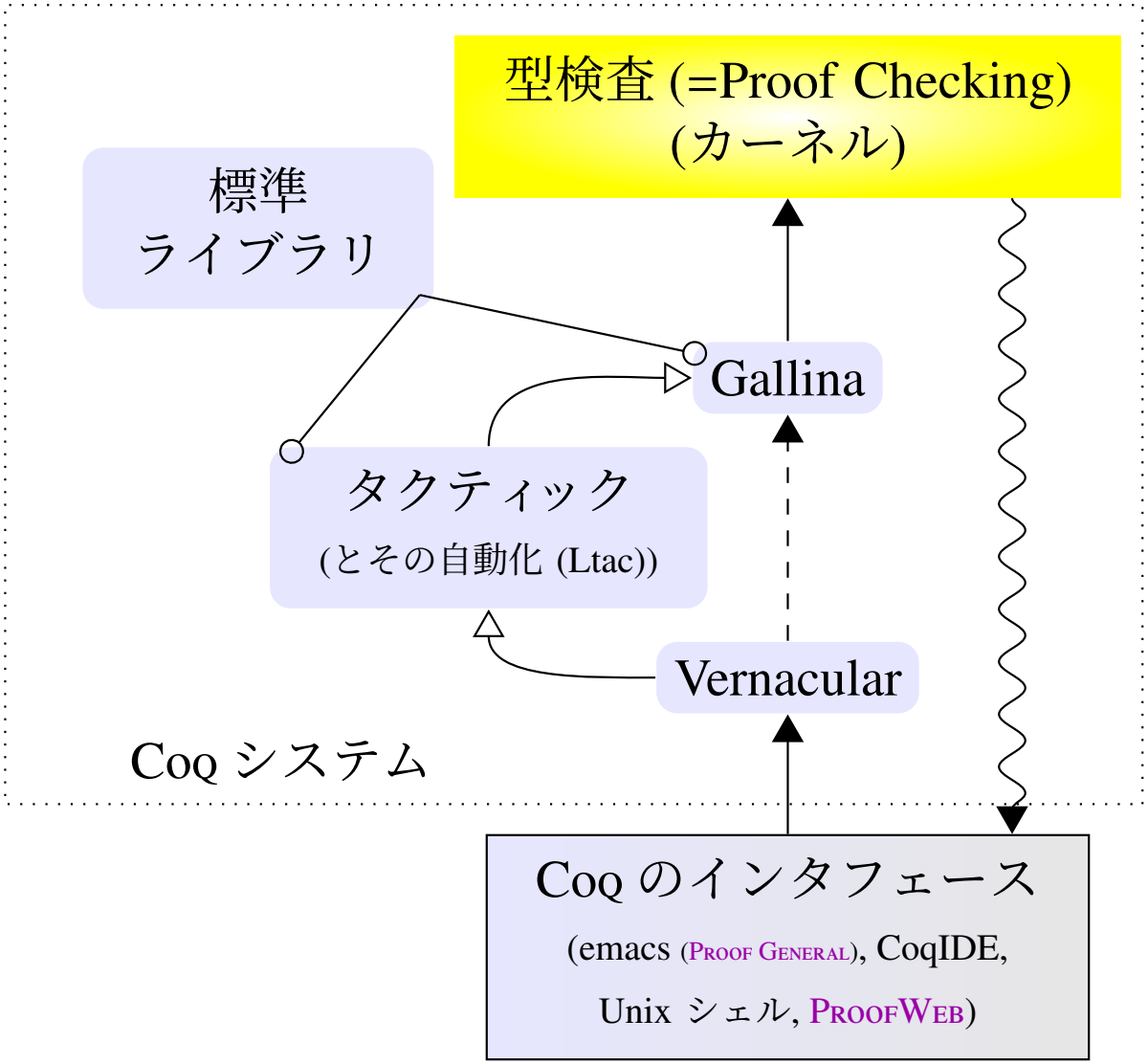
$$\frac{A \to B \qquad A}{B}$$

- プログラミング言語の関数適用:
  f が A→B という型をもち a が A という型を持つ, ならば, $f(a)$($f\ a$ と書く)が B という型を持つ:

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

  ⇒ Modus Ponens に見える ⇒ 含意は関数型
- $H : A$ は「H は A が成り立つという証明である」として解釈する

Coq システムの概要



—▶: Vernacular というコマンド言語で新しいデータ構造や証明を追加する

– –▶: 証明項は Gallina という関数型言語で直接記述できる

—▷: 実際は, 証明項をタクティックで間接に記述する

—○: Coq の標準ライブラリは検証済みの再利用可能なデータ構造や定理やタクティッくを提供する

⇀▶: 記述した証明が型検査が通らない場合, ユーザまでフィードバック

自然演繹による Hilbert の公理 S の証明

自然演繹の一部の規則

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ \text{axiom} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_i \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \to_e$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash A \to B \to C}\ \text{axiom} \quad \dfrac{}{\Gamma \vdash A}\ \text{axiom}}{\Gamma \vdash B \to C}\ \to_e \quad \dfrac{\dfrac{}{\Gamma \vdash A \to B}\ \text{axiom} \quad \dfrac{}{\Gamma \vdash A}\ \text{axiom}}{\Gamma \vdash B}\ \to_e}{\Gamma^1 \vdash C}\ \to_e}{A \to B \to C, A \to B \vdash A \to C}\ \to_i}{A \to B \to C \vdash (A \to B) \to A \to C}\ \to_i}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\ \to_i$$

[1] $\Gamma = A \to B \to C, A \to B, A$

Gallina

- Coq で, 証明項は Gallina という型付きプログラミング言語で記述する
- 項 (NB: 型を含む):

t	:=	x, A	変数
	|	`forall` $x : A, B$ | $A \to B$	product
	|	`Prop` | `Set` | `Type`	ソート
	|	`fun` x => t	関数
	|	`let` x:=t_1 `in` t_2	ローカル定義
	|	$t_1\ t_2$	適用
	|	c	constant
	|	`match` t `with` | *pattern* => t `end`	
	|	`fix`$f\ x : A := t$	匿名の不動点

- $A \to B$ が `forall` $a : A, B\ a$ に一般化される (NB: 返り値の型が入力の値に依存する)
- `fun` は型 `forall` の値, 適用で消費する;
  帰納的に定義された型の値は構成子 (スライド 38) となり, `match` で消費する
- Gallina の実行が停止することを保証する必要がある
  - 簡単なケースではシステムが保証 (`struct` 構文)
  - それ以外は証明が要る (やり方: [BC04, Chapter 15])

型付け規則 (依存型なし)

- Coq のカーネルは次の型 judgment を確認する: $E, \Gamma \vdash t : A$
  - 型検査は決定可能
- Proposition-as-types $\Rightarrow$ t = 証明, A = 言明
  E = グローバル環境, Γ = ローカルコンテキスト
  $E, \Gamma = x_0 : A_0$ (仮定), $x_1 : A_1 := t_1$(定義)
  (NB: これから, E を省略する)
- [CDT12, Sect. 4.2] による:

$$\frac{x : A \in \Gamma \text{ or } \exists t. x : A := t \in \Gamma}{\Gamma \vdash x : A}\ \text{Var} \qquad \text{仮定の利用}$$

$$\frac{\Gamma \vdash A \to B : \begin{cases} \texttt{Prop} \\ \texttt{Set} \\ \texttt{Type}_i \end{cases} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\, x \Rightarrow t : A \to B}\ \text{Lam} \qquad \text{仮定の導入}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\ t_2 : B}\ \text{App} \qquad \text{補題の利用}$$

証明項付きの Hilbert の公理 S の証明 I

1. 最初のゴール: $\vdash ?_0 : (A \to B \to C) \to (A \to B) \to A \to C$
2. Lam を適用すると, ゴールが変わる:
   $H_1 : A \to B \to C \vdash ?_1 : (A \to B) \to A \to C$
   $?_0 = \lambda H_1 : A \to B \to C.?_1$
3. Lam を 2 回適用すると: $\Gamma^2 \vdash ?_3 : C$
   $?_1 = \lambda H_2 : A \to B.\lambda H_3 : A.?_3$
4. App(パラメーター B) を適用すると, 2 つのゴールが発生する:
   - 4.1 (1) $\Gamma \vdash ?_4 : B \to C$
   - 4.2 (2) $\Gamma \vdash ?_5 : B$
   - 4.3 $?_3 = ?_4\ ?_5$
5. (1) に App(パラメーター A) を適用すると, 新たなゴール 2 つが発生する:
   - 5.1 (3) $\Gamma \vdash ?_6 : A \to B \to C$
   - 5.2 (4) $\Gamma \vdash ?_7 : A$
   - 5.3 $?_4 = ?_6\ ?_7$
6. (3) に Var を適用すると, $?_6 = H_1$ と分かる
7. (4) に Var を適用すると, $?_7 = H_3$ と分かる

証明項付きの Hilbert の公理 S の証明 II

8. 上記のプロセスを続けると, 最終に次の証明になる:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash H_1 : A \to B \to C}\,\mathrm{Var} \quad \dfrac{}{\Gamma \vdash H_3 : A}\,\mathrm{Var}}{\Gamma \vdash H_1\ H_3 : B \to C}\,\mathrm{App} \quad \dfrac{\dfrac{}{\Gamma \vdash H_2 : A \to B}\,\mathrm{Var} \quad \dfrac{}{\Gamma \vdash H_3 : A}\,\mathrm{Var}}{\Gamma \vdash H_2\ H_3 : B}\,\mathrm{App}}{\Gamma \vdash (H_1\ H_3)\,(H_2\ H_3) : C}\,\mathrm{App}}{H_1 : A \to B \to C, H_2 : A \to B \vdash \lambda H_3 : A.\ (H_1\ H_3)\,(H_2\ H_3) : A \to C}\,\mathrm{Lam}}{H_1 : A \to B \to C \vdash \lambda H_2 : A \to B.\lambda H_3 : A.\ (H_1\ H_3)\,(H_2\ H_3) : (A \to B) \to A \to C}\,\mathrm{Lam}}{\vdash \lambda H_1 : A \to B \to C.\lambda H_2 : A \to B.\lambda H_3 : A.\ (H_1\ H_3)\,(H_2\ H_3) : (A \to B \to C) \to (A \to B) \to A \to C}\,\mathrm{Lam}$$

⇒ 証明のステップは型検査規則の適用として考えていい
⇒ Coq ではタクティックという命令として実現している
(ただし, タクティックは型検査規則の適用と完全に一致はしない)

[2] $\Gamma = H_1 : A \to B \to C, H_2 : A \to B, H_3 : A$

Vernacular ⇜ スクリプト ⇜ タクティック

- タクティック
  - 証明 (NB: つまり, Gallina の項) をタクティックを用いて対話的に組み立てる
  - カーネルは一段階ずつ確かめる (最終なチェックもある)
- Vernacular 構文 (NB: Gallina ではない):
  - `Lemma`: 証明モードに入る
  - `Qed`: 成功すると, 補題はグローバル環境 E に保管される
  - `Show Proof`: 証明項自体を見せる
- スクリプト $\stackrel{def}{=}$ 連続したタクティック

Hilbert の公理 S の証明のまとめ

ゴール:
$\vdash (A \to B \to C) \to (A \to B) \to A \to C$
使った型付け規則:
Lam
Lam
Lam
App B
App A
Var
Var
App A
Var
Var

Coq スクリプト

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
intro H1.
intro H2.
intro H3.
cut B.
cut A.
assumption.
assumption.
cut A.
assumption.
assumption.
Qed.
```

Curry-Howard 同型対応

- 型付き λ 計算 $\leftrightarrow$ 命題論理: Curry-Tait 同型対応; $\lambda\Pi$ 計算 $\leftrightarrow$ 述語論理: Curry-de Bruijn-Howard 同型対応; ...
- Curry-Howard 同型対応によって, $\bot$ の型を持つ項はないことが分る; しかし, Gallina の場合, その対応は明示的に書いていない [Wie14]
- Coq は consistent である (証明のない型がある): `forall` P : `Prop`, P 型を持つ項はない (強正規化の結果) [Pot03]
- Brouwer-Kolmogorov-Heyting の意味論:

$A \to B$	A の証明を受けて, B の証明を返す関数
`forall` $x : A, B$	A の証明 t を受けて, $B\{t/x\}$ の証明を返す関数
$A \wedge B$	A の証明と B の証明の pair
$\exists x : A.B$	項 t と $B\{t/x\}$ の証明の pair
$A \vee B$	A または B の証明
$\bot$	inhabitant は空

Conversion ルール

- Coq が H. Poincaré 原理を実現している:
  - 「2 + 2 = 4 の証明は計算の問題である」
- Gallina 項の syntactic 同値関係は計算 modulo である

$$\frac{\Gamma \vdash t : A \qquad A =_{\beta\delta\zeta\iota} B}{\Gamma \vdash t : B}\ \text{Conv}$$

(NB: 型検査決定可能なため, 停止性の保証が要る)

- Deduction は計算になる時に, リフレクションという (スライド 68)
- $=_{\beta\delta\zeta\iota}$ [CDT12, Sect. 4.3]:
  - β: β 簡約 (つまり, $(\texttt{fun}\, x \Rightarrow t_1)t_2 \to_\beta t_2\{t_1/x\}$)
  - δ: 定義を展開
  - ζ: `let` の代入
  - ι: 帰納的に定義された型の値の消費

ソート (Sorts)

- Small sorts: `Prop` (命題の型), `Set` (データ構造の型)
  - 例: `nat : Set` (自然数);
    $A \to$ `Prop` という関数型は A 型を持つ 1 つ引数を受け取る述語を表す
  - `Set` のデータ構造は `discriminate` できる ($0 \neq 1$)
  - 一般的に, `Prop` の証明は解析できない
    - A : `Prop` 型を持つ証明を消費して, B : `Set` 型を持つものを作れない
    - 例外: `A /\ B`, `x = y`, `{x | P x}` (`P : A ->Prop`, `A : Prop`) [CDT12, Sect. 4.5.4]
    - $\Rightarrow$ Proof irrelevance は admissible
- Large sorts: $\texttt{Type}_i$ (universe とも呼ぶ)
  - ヒエラルキー:

$$\frac{}{\Gamma \vdash \texttt{Prop} : \texttt{Type}_i} \qquad \frac{}{\Gamma \vdash \texttt{Set} : \texttt{Type}_i} \qquad \frac{i < j}{\Gamma \vdash \texttt{Type}_i : \texttt{Type}_j}$$

  - ユーザは `Type` と書く, Coq がレベルを推論する
    - `Set Printing Universes`
    - 失敗すると, universe inconsistency エラー
- Conversion ルールは universe ヒエラルキーで拡張されている:
  - ~~$\leq_{\beta\delta\zeta\iota}$~~ $\to \leq_{\beta\delta\zeta\iota}$
  - Cumulativity: $\texttt{Prop} \leq_{\beta\delta\zeta\iota} \texttt{Type}_i$; $\texttt{Set} \leq_{\beta\delta\zeta\iota} \texttt{Type}_i$; $\texttt{Type}_i \leq_{\beta\delta\zeta\iota} \texttt{Type}_j, i \leq j$
  - $\texttt{Prop} \leq_{\beta\delta\zeta\iota} \texttt{Set}$; . . . [CDT12, Sect. 4.3]

依存型 (forall $x : A, B$ / $\prod_{x:A} B$) の構成

$$\frac{\Gamma \vdash A : \left\{ \begin{array}{l} \texttt{Set} \\ \texttt{Prop} \end{array} \right. \qquad \Gamma, x : A \vdash B : \texttt{Set}}{\Gamma \vdash \texttt{forall}\, x : A, B : \texttt{Set}}\ \text{Prod-Set}$$

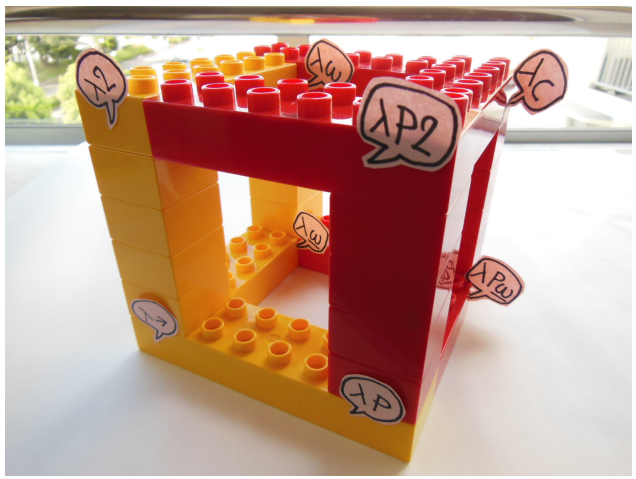
例: `nat -> nat : Set`

$$\frac{\Gamma \vdash A : \left\{ \begin{array}{l} \texttt{Prop} \\ \texttt{Set} \\ \texttt{Type}_i \end{array} \right. \qquad \Gamma, x : A \vdash B : \texttt{Prop}}{\Gamma \vdash \texttt{forall}\, x : A, B : \texttt{Prop}}\ \text{Prod-Prop}$$

例: `(forall x : nat, 0 <= x) : Prop`
`(forall x : nat, x = x) : Set`
`(forall A : Prop, A -> A) : Prop` (polymorphism)

$$\frac{\Gamma \vdash A : \texttt{Type}_{i \leq k} \qquad \Gamma, x : A \vdash B : \texttt{Type}_{j \leq k}}{\Gamma \vdash \texttt{forall}\, x : A, B : \texttt{Type}_k}\ \text{Prod-Type}$$

例: `Prop -> Prop : Type`
`Set -> Set : Type`
`(forall A : Set, A -> A) : Type`
`nat -> Prop : Type` (依存型)
`(Prop -> Prop) -> Prop : Type` (higher-order polymorphism)



Prop impredicative / Type predicative

紙上

- Prop は impredicative: B は Prop なら, A を問わず, forall $x : A, B$ は Prop となる; 例えば [Pot03]:

$$\dfrac{\dfrac{\dots}{\vdash \texttt{Prop} : \texttt{Type}_0} \qquad \dfrac{\dfrac{\dots}{A : \texttt{Prop} \vdash A : \texttt{Prop}} \qquad \dfrac{\dots}{A : \texttt{Prop}, \_ : A \vdash A : \texttt{Prop}}}{A : \texttt{Prop} \vdash A \to A : \texttt{Prop}}}{\vdash \texttt{forall}\, A : \texttt{Prop}, A \to A : \texttt{Prop}}$$

- Type は predicative:

$$\dfrac{\dfrac{\dfrac{\dots}{0 < 1}}{\vdash \texttt{Type}_0 : \texttt{Type}_1, 0 \le 1} \qquad \dfrac{\dfrac{\dots}{A : \texttt{Type}_0 \vdash A : \texttt{Type}_0} \qquad \dfrac{\dots}{A : \texttt{Type}_0 \vdash \texttt{Type}_0 \le_{\beta\delta\zeta\iota} \texttt{Type}_1}}{A : \texttt{Type}_0 \vdash A : \texttt{Type}_1, 1 \le 1}}{\vdash \texttt{forall}\, A : \texttt{Type}_0, A : \texttt{Type}_1}$$

Prop impredicative / Type predicative

Coq で

- 例えば, $\forall P : \texttt{Prop}, P \wedge P$ の証明を考える:

```
Definition DupProp : forall (P : Prop), P -> P /\ P :=
  fun P p => conj p p.
```

DupProp の型は Prop にあるので, Check (DupProp _ DupProp). は成功する

- 一方, $\forall P : \texttt{Type}, P \wedge P$ の証明を考える:

```
Definition DupType : forall (P : Type), P -> P * P :=
  fun P p => (p, p).
```

DupType の型は Type にあるが, Check (DupType _ DupType). は失敗する

- 他の例:

```
Definition myid : forall (A : Type), A -> A :=
  fun (A : Type) (a : A) => a.
```

- Coq 8.5 は universe polymorphism を導入 [ST14]

参考ファイル➡predicative_example.v

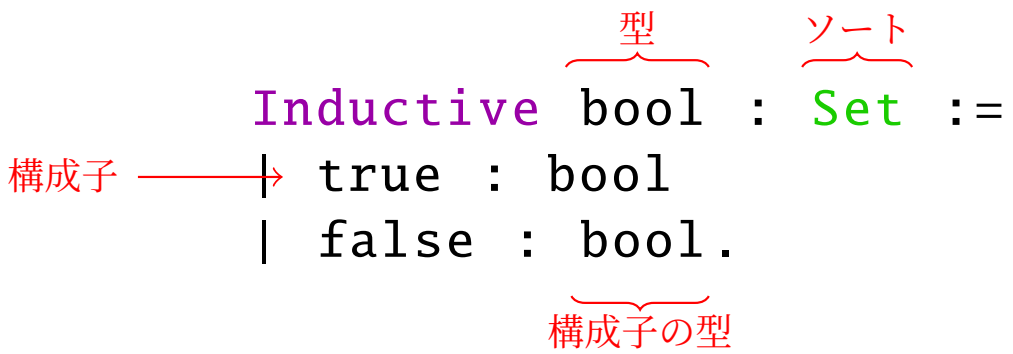
型付け規則 (依存型あり)

$$\frac{\Gamma \vdash \texttt{forall}\, x : A, B : \begin{cases} \texttt{Prop} \\ \texttt{Set} \\ \texttt{Type}_i \end{cases} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\, x \Rightarrow t : \texttt{forall}\, x : A, B}\ \text{Lam}$$

$$\frac{\Gamma \vdash t_1 : \texttt{forall}\, x : A, B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\ t_2 : B\{t_2/x\}}\ \text{App}$$

帰納的に定義される型

enumerated 型



ブール型の定義によって, 次の導入が行われる:

- constants: `bool`, `true`, `false`
- 三つまでの elimination rules:
  - `bool_rect`: strong elimination (NB: `Type` の述語) (次のスライドに参考)
  - `bool_ind`: (NB: `Prop` の述語) → 場合分け (`case` タクティックに参考)
- ι-簡約ルール:

```
match true with true => t1 | false => t2 end ⟶ι t1
match false with true => t1 | false => t2 end ⟶ι t2
```

Elimination ルールによるプログラム

- T 型のデータ構造による場合分け: T_rec または match 構文
- bool_rect (NB: bool_rec, bool_ind はインスタンスである) :

$$\frac{\Gamma \vdash P : \mathtt{bool} \to \mathtt{Type} \qquad \Gamma \vdash t_1 : P\,\mathtt{true}, \Gamma \vdash t_2 : P\,\mathtt{false} \qquad \Gamma \vdash b : \mathtt{bool}}{\Gamma \vdash \mathtt{match}\, b\, \mathtt{with} \mid \mathtt{true} \Rightarrow t_1 \mid \mathtt{false} \Rightarrow t_2\, \mathtt{end} : P\, b}$$

例: ブール型の入力によってリターン型が変る (依存型)

```
Definition typ : bool -> Set :=
  fun b => match b with true => bool | false => nat end.
```

match 構文を利用

```
Definition mkTyp : forall b, typ b :=
  fun b => match b with
             | true => true
             | false => 0
           end.
```

bool_rec を利用

```
Definition mkTyp2 :=
  bool_rec typ true 0.
```

帰納的に定義される型

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

構成子の argument の型

- 自然数 nat の型は Peano 整数を定義する
  - O (大文字の「オー」) は零, S で 1, 2, . . . を作る
- 再帰的な関数の定義のプログラム:
  - Gallina の fix (匿名の不動点) / Vernacular の Fixpoint (named 不動点)
- 数学帰納法による証明
  - 帰納的に定義される型の定義の際, Coq は帰納法の原理を自動生成する
  - elim に参照, スライド 60

数学的帰納法

- 関数の型として言明を記述する:

```
forall P : nat -> Prop,
P 0 ->
(forall n : nat, P n -> P (S n)) ->
forall n : nat, P n
```

  - P : nat ->Prop は命題
  - P 0 は 0 の場合 P が成り立つこと
  - forall P, P n ->P (S n) は帰納ケースのこと
  - forall n, P n は証明したいことのパターン

- 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
  - forall n, P n -> P (S n): 結果の型は入力 n によって異なる
- 自然数を定義する際, Coq は帰納法の定理も裏で証明する:

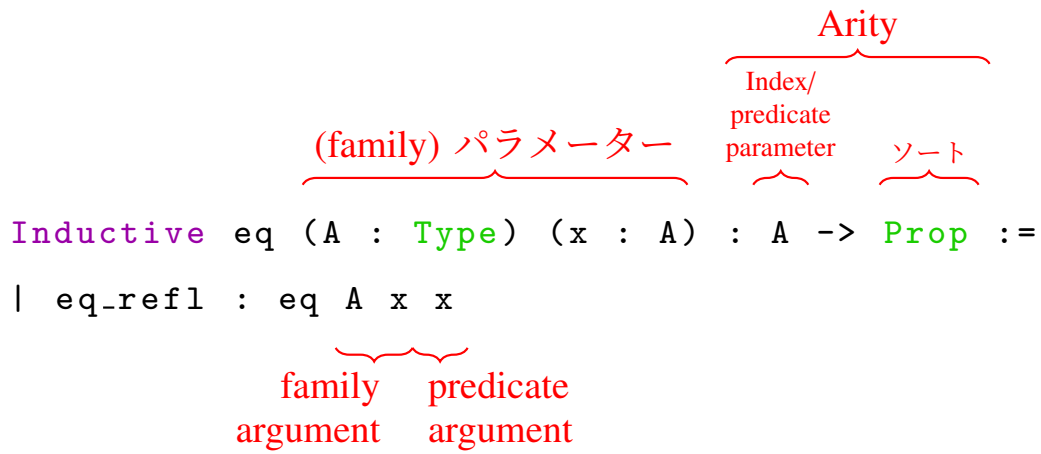
```
Fixpoint nat_ind (P : nat -> Prop) (P0 : P 0)
  (IH : forall n, P n -> P (S n)) (n : nat) :=
 match n with
  | 0 => P0
  | S m => IH m (nat_ind P P0 IH m)
 end.
```

  - nat_ind の実行すると, 0 の場合, P 0 を返す; それ以外の場合, 帰納のケースと再帰関数呼出 (帰納法の仮定に適用) を利用する

帰納的に定義された論理結合子の例

自然演繹	Coq
$\dfrac{}{\Gamma \vdash \mathit{True}}\ \mathrm{True}_i$	`Inductive True : Prop :=` `| I : True.`
~~$\times$~~	`Inductive False : Prop := .`
$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_i$	`Inductive and (A B : Prop) : Prop :=` `| conj : A -> B -> A /\ B.` 例: `conj I I : True /\ True` (NB: `conj p q` $\stackrel{def}{=}$ `@conj P Q p q` $\stackrel{def}{=}$ `@conj _ _ p q`) (NB: `A`と `B`は暗黙のパラメーター, 穴埋め `_`で推論できる) (伝統的な) タクティック: `split`
$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee_{i1}$ $\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee_{i2}$	`Inductive or (A B : Prop) : Prop :=` `| or_introl : A -> A \/ B` `| or_intror : B -> A \/ B` 例: `or_intror False I : False \/ True` (伝統的な) タクティック: `left`, `right`

帰納的に定義される型: 同値関係



- 同値関係 $x =_A y$ は Coq で `x = x` と書ける (A は推論される)
- `eq` は型族を定義する
  - `eq A x x` という型は型 `A` (パラメーター) と
  - 項 `x` (型 `A`) (index) に依存する

  つまり, 同値関係は依存型である
- 同値関係の構成子の型:

```
eq_refl : forall (A : Type) (x : A), x = x
```

Coq の `reflexivity` タクティックは `apply eq_refl` である

Leibniz(/Propositional) Equality

- Leibniz equality は elimination ルール (`eq_rect` 等) の結果
- 書き換えは elimination ルール `eq_ind` によってできる:

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

- `eq_ind` は `eq_rect` のインスタンスである:

$$\frac{\Gamma \vdash A : \mathtt{Type}, x : A, y : A, P : A \to \mathtt{Type} \qquad \Gamma \vdash e : x = y \qquad \Gamma \vdash t : P\,x}{\Gamma \vdash \mathtt{match}\ e\ \mathtt{in}\ \_ = y_0\ \mathtt{return}\ P\,y_0\ \mathtt{with}\ |\ \mathtt{eq\_refl} \Rightarrow t\ \mathtt{end} : P\,y}$$

Dependent Pairs I

- Dependendent pair は存在記号でも書く $(\Sigma_{x:A}.P\,x)$
- ヨ 存在記号:

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
| ex_intro : forall x : A, P x -> exists x, P x
```

タクティック: `exists` $\stackrel{def}{=}$ `apply ex_intro`

- Weak dependent sum, a.k.a. subset type (記号: `{x | P x}`)
  - 例えば, 3 より小さい自然数の集合:

```
{n : nat | n < 3} : Set
```

(Ordinal に参照, スライド 86)

  - ```
    Inductive sig (A : Type) (P : A -> Prop) : Type :=
      exist : forall x : A, P x -> {x | P x}
    (* projections: proj1_sig, proj2_sig *)
    ```
    ```

Dependent Pairs II

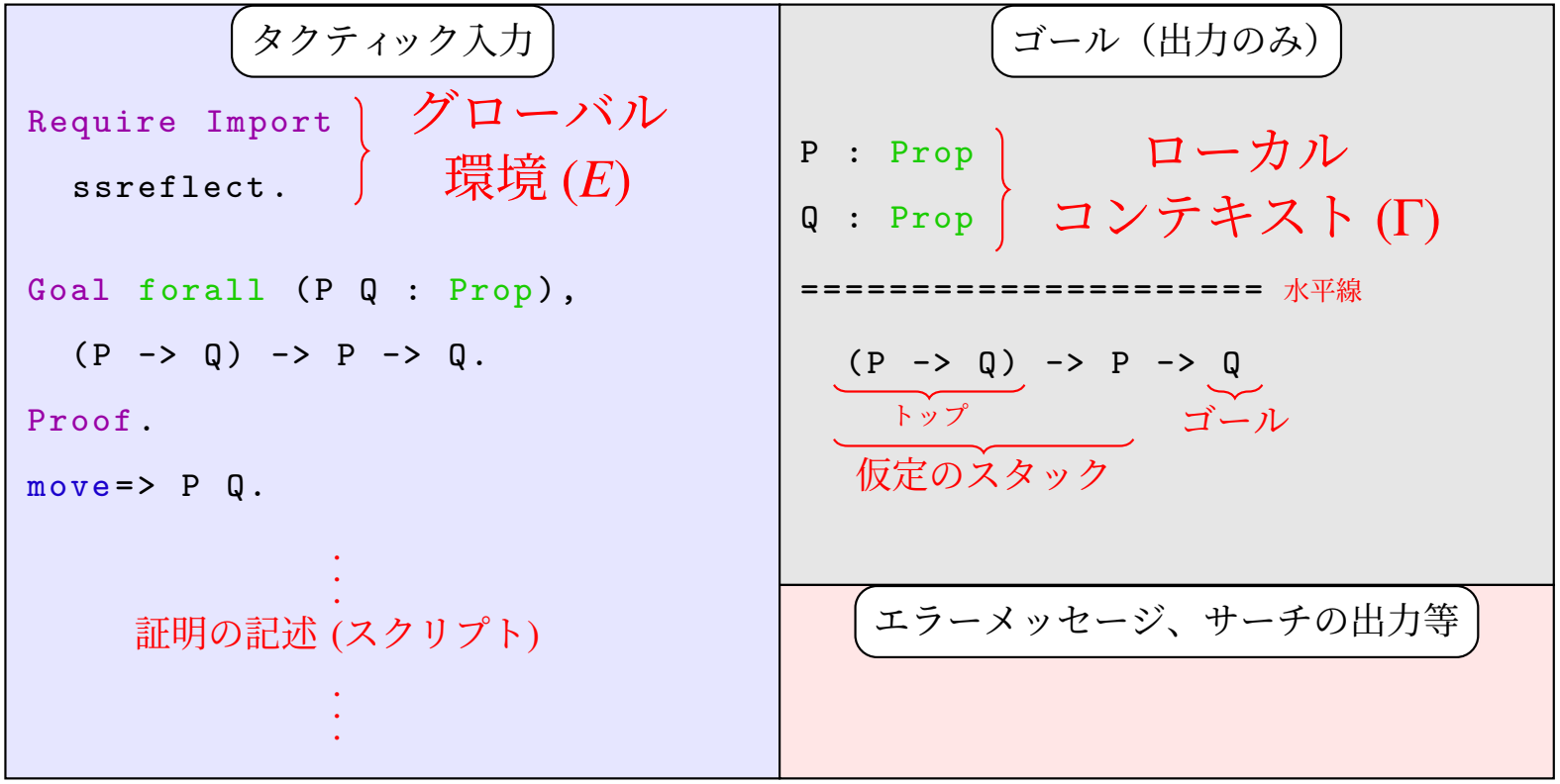
- Dependent record は dependent pair の一般化:

```
Record sig (A : Type) (P : A -> Prop) : Type :=
  exist { witness : A ; Hwitness : P witness }.
```

- Strong dependent sum:

```
Inductive sigT (A : Type) (P : A -> Type) : Type :=
    existT : forall x : A, P x -> sigT P
(* projections: projT1, projT2;
   injection: Eqdep.EqdepTheory.inj_pair2 *)
```

Coq のインタフェース (Proof General[3], CoqIDE)



(NB: SSReflect のタクティックにとってトップが特別な役割を果たすことが多い)

[3] M-x proof-display-three-b

Coq の標準タクティックについて

標準タクティックが多い

- Coq の標準タクティックは 100 個以上がある
  - マニュアルの 8 章, タクティックの修飾子 (dependent, using, with, at, simple, functional 等のキーワード, cbv のフラッグ等) を除いて
- タクティカル (タクティックの組み合わせ) もある
- 増える一方, 冗長性がある, 一定ではない

ただし, 上記の状況は欠点と見做すべきではない!

NB: Vernacular のコマンドは 120 個以上がある

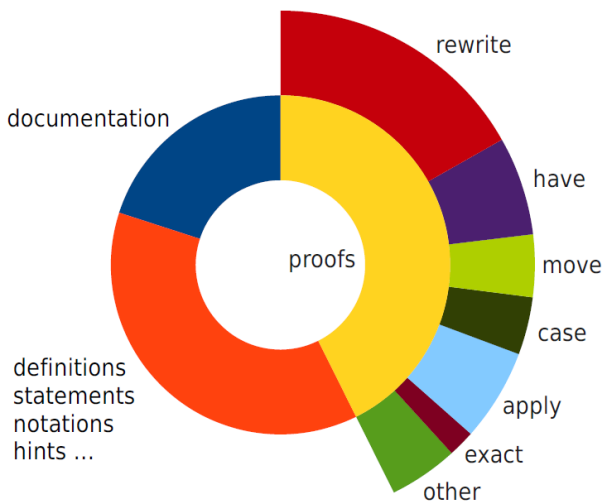
- Transparent/Opaque, Set/Unset 等の修飾子を除いて, 全 Printing オプションを含まない

Coq と SSReflect の関係

- SSReflect は Coq の拡張である:
  - 四色定理の形式化の際, Coq のパージング機能によって拡張 [Gon05]
  - 現在, plug-in の形 (Coq のカーネルの変更なし)
  - HOL に基づく SSReflect もある [Hal14]
- Gallina と Vernacular は殆ど変更なし
  - pattern testing: `if t1 then pattern then t2 else t3`
  - destructuring assignment: `let:` (Coq の `let` と `let '` の一般化)
- コアタクティックの向上
  - 簡単化と一般化, 小規模リフレクション (スライド 68)
  - スクリプトの構造化, 名付けることの強制
  - スクリプトと証明は短くなる
- 新しいライブラリ (四色定理, 奇数位数定理から来る)
  - 標準タクティックと標準ライブラリはまだ使える

一番使われているタクティック?

- 四色定理のスクリプトで ([Gon05] 85%のタクティック [4]: `Move`, `Case`, `Apply`, `Step`[5], `Rewrite` (`Move+Case` と同じぐらい多い)
- [GM10] による:
  - 1/3: bookkeeping (仮定に名付け, 要らなくなった仮定の削除, `move`, `have`)
  - 1/3: 書き換え (`rewrite`)
  - 1/3: deduction (`apply`, `case`, `exact`)
- MathComp のスクリプトの中で (2012 年の夏のころ) (図: Enrico Tassi の ITP2012 のスライドから, [GT12] にも参照)



[4] Coqv.7 のころ, タクティック名の先頭は大文字だった
[5] 今の `have`

move タクティック

- 役割 1: 仮定の導入
- move=>H. はトップをローカルコンテキストにポップしてから, H と名付ける (NB: トップ $\stackrel{def}{=}$ 水平線の一番左にある仮定)

ゴール (前)	タクティック	ゴール (後)
``======================``` ```forall P Q : Prop,``` ```(P -> Q) -> P -> Q```	move=>P.	```P : Prop``` ```======================``` ```forall Q : Prop,``` ```(P -> Q) -> P -> Q```

- Coq の intros, clear, pattern を一般化する (NB: タクティカルとの組み合わせ, これから説明する)

参考ファイル➡tactics_example.v

move=>タクティックによる証明

move=>は $\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\, x \Rightarrow t : \texttt{forall}\, x : A, B}$ Lam (simplified) に当たる:

ゴール（前）	タクティック	ゴール（後）
`======================` `forall P Q : Prop,` `(P -> Q) -> P -> Q`	`move=>P.`	`P : Prop` `======================` `forall Q : Prop,` `(P -> Q) -> P -> Q`
証明（前） `?1`		証明 (後) `(fun P : Prop => ?2)`

move:タクティック (discharge)

move: H. はローカルコンテキストから仮定 H をスタックのトップにプッシュする:

ゴール (前)	タクティック	ゴール (後)
P : Prop ===== forall Q : Prop, (P -> Q) -> P -> Q	move: P.	===== forall P Q : Prop, (P -> Q) -> P -> Q

- move: (H). はローカルコンテキストから H を消さない
- グローバル環境からもプッシュできる
  - move: (lem a b). が (lem a b) の結論の型を, 仮定として, プッシュする
- :と=>はタクティカルという (他のタクティックと組合せて使う)
- Coq の generalize, revert に当る

apply タクティック

- `apply.` はトップがゴールを導くかどうかを単一化を用いて確認し, 適用する (NB: ゴールは水平線の一番右にある型):
- `apply: H.` $\stackrel{def}{=}$ `move: H. apply.` (NB: プッシュしてから, タクティックを実行)

ゴール (前)	タクティック	ゴール (後)
`P : Prop` `Q : Prop` `PQ : P -> Q` `p : P` `======================` `Q`	`apply: PQ.`	`P : Prop` `Q : Prop` `PQ : P -> Q` `p : P` `======================` `P`

- `apply =>H.` $\stackrel{def}{=}$ タクティックを実行してから, ポップする
- backward reasoning の主なタクティック
- 仮定が足りなければ, ユーザにサブゴールが求められる
- 実際に, `apply: H.` は Coq の `refine (H _ ... _).` を一般化 (NB: _の数は自動調整)

case タクティック

case はトップが帰納的に定義されていたら, どの構成子でできているのか順に場合分けをし, サブゴールを生成する; 例えば:

ゴール (前)	タクティック	ゴール (後)
P : Prop Q : Prop ===== P /\ Q -> Q /\ P	case.	P : Prop Q : Prop ===== P -> Q -> Q /\ P

- case: H. $\stackrel{def}{=}$ move: H. case.
- case は Coq の destruct, injection を一般化する
- case は move=>[] と書ける
- 複数のゴールは求められる時に, case=>[H1 |H2] と書く

case による証明

ゴール (前)	タクティック	ゴール (後)
P : Prop Q : Prop ===== P /\ Q -> Q /\ P	case.	P : Prop Q : Prop ===== P -> Q -> Q /\ P
証明 (前)		**証明 (後)**[6]
(fun P Q : Prop => ?3)		(fun (P Q : Prop) (H : P /\ Q) => match H with | conj H0 H1 => ?10 H0 H1 end)

[6] simplified

rewrite タクティック

rewrite は一つのマッチパターンを見つけて, 全ての出現を書き換える;
例えば, ゴールの中の n (等式 n0 の左辺) を 0 (右辺) で置き換える:

ゴール (前)	タクティック	ゴール (後)
n : nat n0 : n = 0 m : nat ===== m + n = m	rewrite n0.	n : nat n0 : n = 0 m : nat ===== m + 0 = m

- 計算を配慮して, マッチを行う; パターンのサーチは外から中へ, 左から右へ
- Coq の rewrite, unfold/fold (NB: 定義の展開), simpl の一般化
- 同値関係の elimination ルールを利用 (NB: rewrite H は apply (eq_ind _ ... _) として考えていい)

rewrite タクティックによる証明

ゴール（前）

```
n : nat
n0 : n = 0
m : nat
======================
 m + n = m
```

タクティック

```
rewrite n0.
```

ゴール（後）

```
n : nat
n0 : n = 0
m : nat
======================
 m + 0 = m
```

証明（前）

```
(fun (n : nat)
     (n0 : n = 0)
     (m : nat) => ?7)
```

証明 (後)[7]

```
(fun (n : nat)
     (n0 : n = 0)
     (m : nat) =>
 eq_ind_r (fun n1 : nat
   => m + n1 = m)
  ?8 n0)
```

[7] simplified

rewrite タクティック—その他の機能

- コンパクトな記述:
  - `rewrite H1 H2.` $\stackrel{def}{=}$ `rewrite H1; rewrite H2.`
  - `rewrite -H.` は逆の書き換え
  - `rewrite !H.` $\stackrel{def}{=}$ `repeat (rewrite H)`
  - `rewrite (_ : lhs =rhs).` は `cutrewrite` の一般化
- 他のタクティックとの組み合わせ:
  - `rewrite /id.` は定義の展開を行う (逆: `rewrite -/id`)
  - `rewrite //.` $\stackrel{def}{=}$ `try done.`
  - `rewrite /=.` $\stackrel{def}{=}$ `simpl.` (NB: `//=` $\stackrel{def}{=}$ `// /=`)
  - `rewrite {H}.` $\stackrel{def}{=}$ `clear H.`
  - `move=>->.` $\stackrel{def}{=}$ `intro TMP; rewrite TMP; clear TMP.`
- コンテキスト指定:
  - `rewrite [H1]H.` はパターン H1 の中を書き換える
  - `rewrite [H1]lock.`: パターン H1 をロックして, 今後 conversion で変らないようする
  - `rewrite [in X in ...X...]H.` (contextual pattern): ゴールを`...X...` とマッチして, その X の中を書き換える
    - `rewrite {1}H.` より丈夫

elim タクティック—帰納法

elim がトップにある全称量化された変数の型 (例えば, T 型) を見て, ゴールのソートを見て (例えば, Prop ソート), 帰納法を行う (この場合, T_ind を用いて):

ゴール (前)	タクティック	ゴール (後)
`======================` `forall n : nat,` `n + n = 2 * n`	`elim.`	`======================` `0 + 0 = 2 * 0` `======================` `forall n : nat,` `n + n = 2 * n ->` } inductive hypothesis `n.+1 + n.+1 = 2 * n.+1`

- つまり, elim は apply: T_ind の拡張
- タクティカル=>との使い方: elim=>[| x IH].
- ユーザが帰納法の補題を選べる (書き方: elim/myT_ind) (NB: 必要な時: mutual inductive types, nested types)

elim タクティックによる証明

ゴール（前）	タクティック	ゴール (後)
`==========` `forall n : nat,` `n + n = 2 * n`		`==========` `0 + 0 = 2 * 0` `==========` `forall n : nat,` `n + n = 2 * n ->` `n.+1 + n.+1 = 2 * n.+1`
証明 (前) `?2`	`elim.`	証明 (後)[8] `(nat_ind` `(fun n : nat =>` `n + n = 2 * n)` `?3 ?4)`

[8] simplified

等式の生成

move H : (t) => h. は仮定 H : h = t を導入する (スタックの中の t を h で書き換える):

ゴール (前)	タクティック	ゴール (後)
s1 : seq nat ===== forall s2 : seq nat, rev (s1 ++ s2) = rev s2 ++ rev s1	move H : (size s1) =>n.	s1 : seq nat n : nat H : size s1 = n ===== forall s2 : seq nat, rev (s1 ++ s2) = rev s2 ++ rev s1

- 入れ子のデータ構造を case する前に役に立つ
- case H : t. は, t が帰納的に定義された型なら, 仮定 H で構成子の情報を記録する
- Coq の case_eq に当たる

congr タクティック

f_equal を一般化と最適化:

ゴール (前)	タクティック	ゴール (後)
`` ====================== a + b + c = a' + b' + c' ```	``` congr (_ + _ + _). ```	``` ====================== a = a' ====================== b = b' ====================== c = c' ```

have と suff タクティック

- forward reasoning $\stackrel{def}{=}$ コンテキストに加えたい仮定を明確にする
  - 紙上の証明と同じ
  - 定理証明支援系 Mizar のような記述 (declarative style と言う)
- `have : t.` は新しいサブゴールを開いて, その証明を求める
  - `have {H}H : t.` $\stackrel{def}{=}$ `t` を証明したら, `move=>{H}H.` を行う
  - `have [x Hx] : t.` $\stackrel{def}{=}$ `t` を証明したら, `case=>x Hx.` を行う
- `suff : t.` は新しい仮定をコンテキストに加えて, その証明は後で求める
- `assert` を一般化
- 関連するタクティック: `wlog`

Structured スクリプト

証明が大きくなると, メンテナンスは重要になる (NB: メンテナンスの時間は証明の記述の二倍になると言われる) ⇒ スクリプトを壊れにくくすることは重要

- スクリプト $\stackrel{def}{=}$ backward reasoning を挟んだ forward reasoning として考えていい
- スクリプトの構造:
  - +, -, *: スクリプトの木構造を明かにする, 普通は三段階まで [GM10]
  - インデント (スペースは二つ): 残るゴールの数を表す
  - 葉: ターミネーターで終了
    - ターミネーター $\stackrel{def}{=}$ 必ず成功するタクティック
      (例えば, `discriminate`, `contradiction`, `assumption`, `exact`, `done`)
    - `by` タクティカルによって, 任意のタクティックがターミネーターになる
      (NB: 四色定理の 25%の `have` は `by` 証明を持つ [Gon05])
  - サブゴールの選択子:
    `last`, `first` を使って, 生成されるサブゴールの順番を変える
  - コピペーしない, その代わりに `set` タクティックを使う (contextual pattern あり [GMT08, Sect. 8.3.1])
- 仮定の名前は自動的に決めない (特に, Coq の `intros`, `induction` 等に任せない)
- 証明ステップ: ~~一つのタクティック~~ → 一行のスクリプト

move + ビュー (view)

move/H. はトップ仮定を変形する

ゴール (前)	タクティック	ゴール (後)
P : Prop Q : Prop PQ : P -> Q ====================== P -> Q	move/PQ.	P : Prop Q : Prop PQ : P -> Q ====================== Q -> Q

- move/PQ $\stackrel{def}{=}$ move=>tmp. move: (PQ tmp). move=>{tmp}.
- 「move=>」とビューの組み合わせの例: move=>P Q PQ /PQ.
- ビューを使うと, ビューヒントによって, ビュー補題は適用される; 例えば, 等価性 PQ : P <->Q がある場合, move/PQ = move/(iffLR PQ) (NB: iffLR : forall P Q : Prop, (P <->Q) ->P ->Q)
- move/(_ a b c) はトップを特化する (Coq の specialize に当る)

参考ファイル➡view_example.v

apply + ビュー

apply/H. はゴールを変形する:

ゴール (前)	タクティック	ゴール (後)
P : Prop Q : Prop PQ : P <-> Q p : P ===== Q	apply/PQ.	P : Prop Q : Prop PQ : P <-> Q p : P ===== P

- 必要であれば, ビューヒントを利用; 例えば, 上記の apply/PQ. は apply/(iffRL PQ). である
- apply H. は apply/H. と書ける ⇒ 連続で使える: apply/H1/H2.

リフレクション

- リフレクションとは? 基本的に, ゴールを証明するために, タクティックを使う代わりに, Gallina の関数 (≃ decision procedure) に任せる
  - 具体的に, `forall a, f a =true ->P a` を満たす `f` 関数があれば, ゴール `P a : Prop` の証明は `f` の実行に当る [Bou97]
  - 例えば, Coq の `ring` タクティックはリフレクションで実装されている
- 利点:
  - 速さ: 証明はカーネル内の計算となる (conversion ルールによる, スライド 32)
  - 証明のサイズ: 証明項は同値関係の証明となる
- 小規模リフレクションとは? 小さなサブゴールでもリフレクションを使って, タクティックによる deduction と計算を繰り返しながら, 対話な証明を続ける
  - `Prop` を `bool` にリフレクションすると, 真理値表を使った証明ができ (`P && Q ==>Q && P` と `P /\Q ->Q /\P` の違い), かつ, 書き換えもできる
  - ブール論理に落とす ⇒ 単純な計算で証明負担を減らし, 書き換えができる
  - 実現のための重要な材料: `is_true` コアーション, `reflect` 補題 (`ssrbool.v`, スライド 78)

ビューとブールリフレクション

and/&&(ブールの世界) と/\(Prop の世界) のとの等価性:

```
andP : forall b1 b2 : bool, reflect (b1 /\ b2) (b1 && b2)
```

ゴール（前）	タクティック	ゴール（後）
P : bool Q : bool ===== P && Q -> Q	move/andP.	P : bool Q : bool ===== P /\ Q -> Q

- 知らなくてもいい: move/andP $\stackrel{def}{=}$ move/(elimTF andP):

```
elimTF : forall (P : Prop) (b c : bool),
  reflect P b -> b = c -> if c then P else ~ P
```

ビューとブールリフレクション + case

case を行う前に, ビューを適用:

ゴール (前)	タクティック	ゴール (後)
P : bool Q : bool ===== P && Q -> Q	case/andP.	P : bool Q : bool ===== P -> Q -> Q

- case/andP $\stackrel{def}{=}$ move/andP. case.
- ビューと「=>」の組み合わせ:
  case/andP =>P Q ↔ move/andP =>[] P Q

ビューとブールリフレクション + case

複数のサブゴールの生成の場合:

ゴール (前)	タクティック	ゴール (後)
P : bool Q : bool ===== P || Q -> P || Q	case/orP.	P : bool Q : bool ===== P -> Q || P ===== Q -> Q || P

- それぞれのサブゴールの仮定を名づける: case/orP =>[H1 |H2].

同値関係とリフレクション

論理演算のように, Prop の同値関係とそのブール版のリフレクションも使う:

ゴール (前)	タクティック	ゴール (後)
`n : nat` `m : nat` `=======` `eqn n m -> n = m`	`move/eqnP.`	`n : nat` `m : nat` `=======` `n = m -> n = m`

- Prop より, bool が便利な場合は, しばしばある
- 多重定義の紹介の時に, その便利さはさらに明確になる (スライド 81)

SSReflect タクティックの纏め

Coq と比較

Coq	SSReflect
intro intros	move=>
revert	move:
generalize	move: (lem a)
specialize	move/(_ x)
rewrite	rewrite move=>->
rewrite <-	rewrite - move=><-
unfold	rewrite /
fold	rewrite -/
cutrewrite	rewrite (_ : a =b)
destruct injection	case
case_eq	case H :

Coq	SSReflect
apply refine	apply:
exact	exact:
assert	have suff
simpl	/=フラッグ
clear H	{H}
elim induction	elim
now	by
discriminate assumption contradiction	done / //フラッグ
pattern	contextual pattern 等
f_equal	congr

Outline

定理証明支援系とは
- 定理証明支援系の概要
- 定理証明支援系の歴史
- 定理証明支援系の応用例
  - 数学の証明の形式化
  - ソフトウェアの形式検証

定理証明支援系 Coq/SSReflect の入門
- Coq による形式証明の原理
- 形式モデルと仕様の作成
- Coq と SSReflect の関係
- 形式証明の基本

SSReflect ライブラリの紹介
- SSReflect ライブラリの概要
- 基礎ライブラリ
- 総和と総乗
- 群と代数

関連するトピックス
- 命令型プログラムの検証
  - アセンブリ言語への応用
  - C 言語への応用
- 数学の証明の形式化
  - 情報理論の形式化
  - 符号理論の形式化

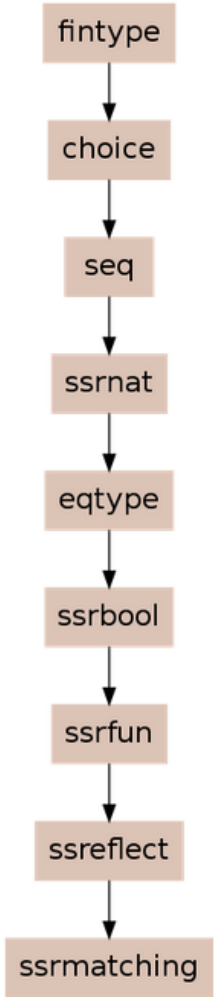
SSReflect ライブラリの概要

- 行数, ファイル (v1.5):
  - 新タクティック: OCaml (約 7,000 行)
  - SSReflect ライブラリ: 約 11,000 行, 9 ファイル
  - MathComp ライブラリ: 約 78,000 行, 53 ファイル
- 一定: 記号, 暗黙の引数の使いかた, コアーション等
  - 記号, 暗黙の引数, コアーション等が分らなくなる時:

`Locate "kigou".`	記号に当たる定義を探す
`Set/Unset Printing Implicit`	暗黙の引数を見せる/見せない
`Set/Unset Printing Notations`	記号を使う/使わない
`Set/Unset Printing Coercions`	コアーションを見せる/見せない
`Set/Unset Printing All`	全部見せる/デフォルトに戻る
`About`	`Check`の変わりに

  - ルールに従わないと大変になる (`Require Import` の順番は大事, 定義の展開はいいアイデアではない)
- ブールが大事 (リフレクション等)
- 副作用を避けるように, SSReflect のライブラリでは, conversion によって, 仮定とゴールが変形しないことになっている

基礎ライブラリ



図は http://ssr2.msr-inria.inria.fr/doc/ssreflect-1.4/ より

ssrfun.v

- ライブラリにおける関数に関する基本的な定義と記号

ssrfun.v notations

`f ^~ y`	`fun x => f x y`
`p .1`	`fst p`
`p .2`	`snd p`
`f =1 g`	`f x = g x`
`{morph f : x / a >-> r}`	`f (aF x) = rF (f x)`
`{morph f : x y / a >-> r}`	`f (aOp x y) = rOp (f x) (f y)`

ssrfun.v definitions

`cancel f g`	`g (f x) = x`
`involutive f`	`cancel f f`
`left_id e op`	`op e x = x`
`right_id e op`	`op x e = x`
`left_zero z op`	`op z x = z`
`right_commutative op`	`op (op x y) z = op (op x z) y`
`right_zero z op`	`op x z = z`
`left_commutative op`	`op x (op y z) = op y (op x z)`
`left_distributive op add`	`op (add x y) z = add (op x z) (op y z)`
`right_distributive op add`	`op x (add y z) = add (op x y) (op x z)`
`self_inverse e op`	`op x x = e`
`commutative op`	`op x y = op y x`
`associative op`	`op x (op y z) = op (op x y) z`

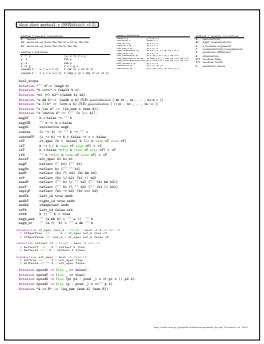
ssrbool.v の重要性

- 規則的な命名規則
  - 一部 ssrfun.v から来る
  - 他のライブラリファイルは似た命名規則を使う
- ブールリフレクションの実現
  - ブール論理の記号と補題
  - is_true コアーション: ブール値から Prop への押し込み, bool は Prop に見える

```
Coercion is_true : bool >-> Sortclass.
```

  - reflect 述語: ブールの世界と Prop の世界の等価性
- pred 述語
  - comprehension による定義によく使う: 有限集合等

参考資料: ssrbool_doc.pdf



例: ssrbool.v の boolP

- `case H : p` は `H : p = true` と `H : p = false` 仮定を生成する (スライド 62)
- ライブラリを効率的に利用できるよう, 記号 (!=等) を使いたい
- 例えば (ここで, eqtype.v(スライド 81) と ssrnat.v を使う (スライド 82)):

ゴール (前)	タクティック	ゴール (後)
`n : nat` `======================` `n * n - 1 < n ^ n`	`case: (boolP (n == 0)).`	`n : nat` `======================` `n == 0 ->` `n * n - 1 < n ^ n` `======================` `n != 0 ->` `n * n - 1 < n ^ n`

参考ファイル ssrbool_example.v

例: ssrbool.v の ifP

- if 文の条件がブール型なら, case: ifP は二つのサブゴールを生成し, 条件は仮定になり, if 文が消える
- 例えば (ここで, ssrnat.v を使う スライド 82):

ゴール (前)	タクティック	ゴール (後)
n : nat ======= odd (if odd n then n else n.+1)	case: ifP.	n : nat ======= odd n -> odd n ======= odd n = false -> odd n.+1

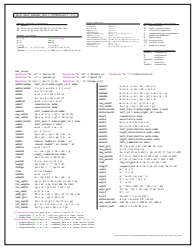
参考ファイル ➡ssrbool_example.v

eqtype.v: 決定可能な同値関係

- 同値関係が決定可能なら, ブール値等式として定義ができる
  - 例えば, 自然数の eqn (スライド 72)
- そのブール値等式と Leibniz 同値関係の等価性が証明できれば, その型は eqType として登録できる
  - reflect 補題を利用
  - 参考ファイル ➡eqtype_example.v
- 利点: canonical structure による多重定義 [MT13]
  - eqType なら, ブール値等式は「==」,「!=」と書ける (NB:「==」は eq_op の記号)
  - 述語の定義；例えば, pred1 a $\stackrel{def}{=}$ [pred x |x == a]
- 書き換え?
  - ==の仮定の書き換え: rewrite (eqP H)
  - reflect 命題を利用: move/eqP で, Leibniz 同値関係とブール値等式を変換

ssrnat.v

参考資料: ssrnat_doc.pdf



- よく考えられた記号
  - 例えば, `Notation "n .+1":= (S n).`
  - eqType として登録済 (eqn, スライド 72)
  - 一定の命名規則
- 定義済のデータ構造と関数の再利用は優先; 例えば:
  - 「<=」は引き算を用いて定義する:

```
leq = fun m n : nat => m - n == 0 : nat -> nat -> bool
```

  - 「<」は「<=」を用いて定義する: `Notation "m < n" := (m.+1 <= n).`
- conversion ルールによって, 変形はしないように, 算術演算はロックされている
  - 例えば, 自然数の加法:

```
Definition addn := nosimpl plus.
Notation "m + n" := (addn m n).
Lemma plusE : plus = addn.
```

ssrnat.v の定義による簡単な補題

例:自然数の不等号「<=」

- Coq の標準ライブラリは帰納的な述語として定義:

```
Inductive le (n : nat) : nat -> Prop :=
    le_n : (n <= n)%coq_nat
  | le_S : forall m : nat, (n <= m)%coq_nat -> (n <= m.+1)%coq_nat
```

- SSReflect では:

```
leq = fun m n : nat => m - n == 0 : nat -> nat -> bool
```

- 補題の例 ([Tas14, Mah14] 等による):

```
Goal forall n, 0 <= n. done. Qed.
Goal forall n m, n.+1 <= m.+1 -> n <= m. done. Qed.
Goal forall n, n <= n. done. Qed.
Goal forall n, n <= n.+1. done. Qed.
Goal forall n, n < n = false. by elim. Qed.
```

- 一般的に, ssrnat.v には one-liner は多い

参考ファイル➡ssrnat_example.v

ssrnat.v: 補題のデザイン

- 書き換えができるように, 等価性「<->」より, 同値関係「=」を利用; 例えば:

```
leq_eqVlt : forall m n : nat, (m <= n) = (m == n) || (m < n)
```

- パラメーター付帰納型族を利用 ⇒ case の際, ゴールの中に現れる関係する項を書き換える; 例えば:
  - Coq の標準ライブラリ:

```
Compare_dec.le_gt_dec :
  forall n m : nat, {(n <= m)%coq_nat} + {(n > m)%coq_nat}
```

  - SSReflect では:

```
leqP : forall m n : nat, leq_xor_gtn m n (m <= n) (n < m)


CoInductive leq_xor_gtn (m n : nat) : bool -> bool -> Set :=
    LeqNotGtn : m <= n -> leq_xor_gtn m n true false
  | GtnNotLeq : n < m -> leq_xor_gtn m n false true
```

参考ファイル➡ssrnat_example.v

seq.v: SSReflect のリスト

seq.v は整理された Coq の標準 list

- seq は list のための記号
  - 例えば, 自然数のリストの型: seq nat
- よく使われる seq 関係の記号:

[::]	$\stackrel{def}{=}$	nil
[:: a; b; c]	$\stackrel{def}{=}$	a :: b :: c :: nil
[seq E |x <- s]	$\stackrel{def}{=}$	map (fun x => E) s
[seq x <- s | C]	$\stackrel{def}{=}$	filter (fun x =>C) s

- s : seq A なら (A : eqType), a \in s は書ける
  - ssrbool.v は predType 型のための共通記号「\in」を定義する
  - mem_seq を用いて, seq を predType として登録
  - 使用例:

```
Variables (T : eqType) (a : pred T).
Fixpoint all s := if s is x :: s' then a x && all s' else true.
Lemma allP s : reflect (forall x, x \in s -> a x) (all a s).
```

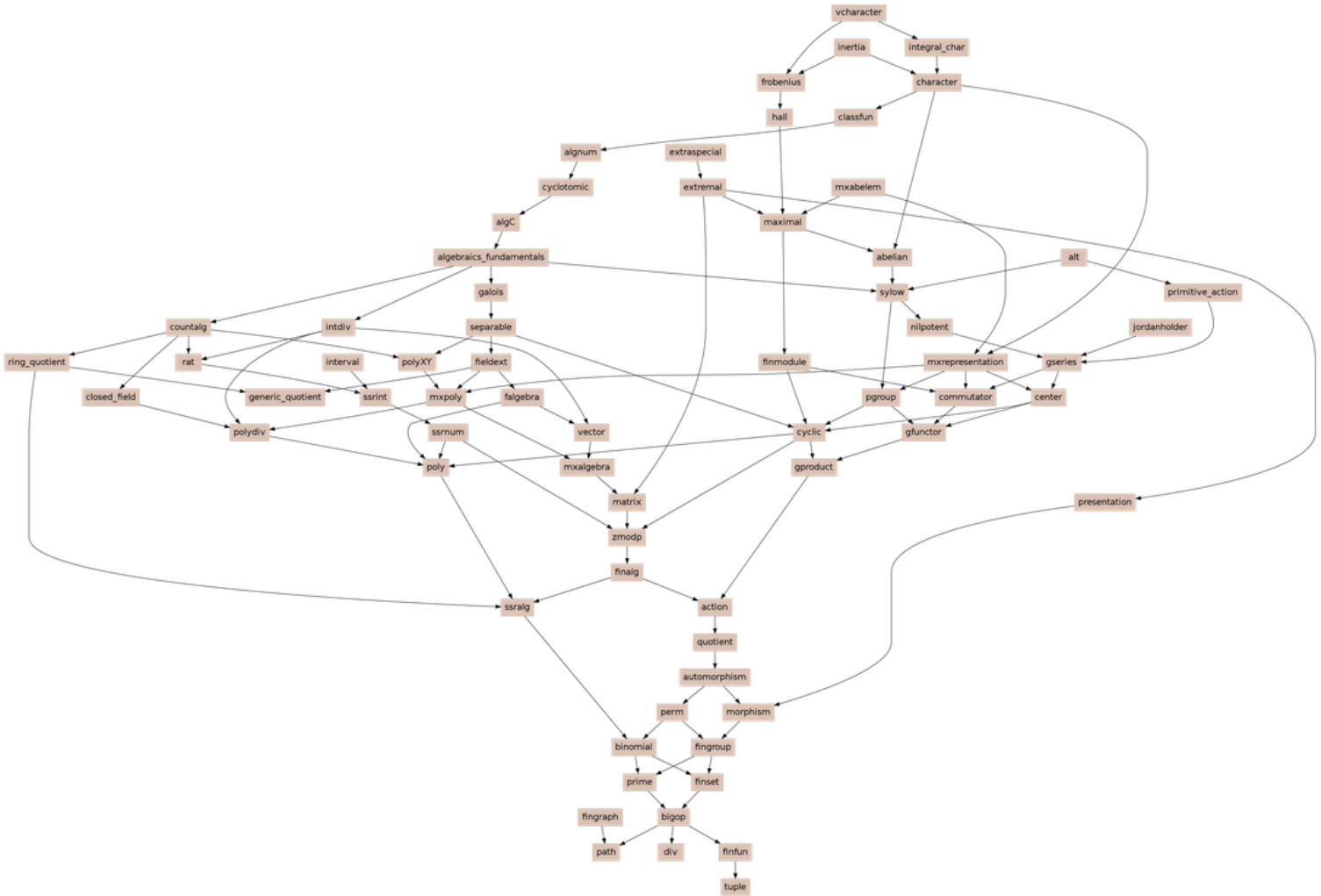
  - 「\in」は有限集合でも使える (スライド 91)

fintype.v

- finType: 有限な数の要素のある型
  - T : finType なら, T の要素のリストは enum T と書く
  - P : pred T なら (T : finType), P を満す要素のリストは enum P と書く
- 代表的な finType: 'I_n (n より小さい自然数)
  - 使用例: 行列の index
  - 定義: Inductive ordinal (n : nat) := Ordinal m of m < n.
  - $\{x \mid P\,x\}$ のような依存型
  - P はブール値なので, 同値関係の証明があれば, 1 つしかない:
    forall (bool : Type) (x y : bool) (p1 p2 : x =y), p1 = p2
    (NB: Coq で同値関係は決定可能でないと証明できない)
  - 従って, 二つの ordinal の比較は自然数の比較と一緒
    - subType による val_inj 単射
- finType だと, 抽象的なアルゴリズムは記述しやすくなる:
  - 全称記号: [forall x, P] (ビュー: forallP)
  - 存在記号: [exists x, P] (ビュー: existsP)
  - 選択: [pick x | P] (仕様: pickP)

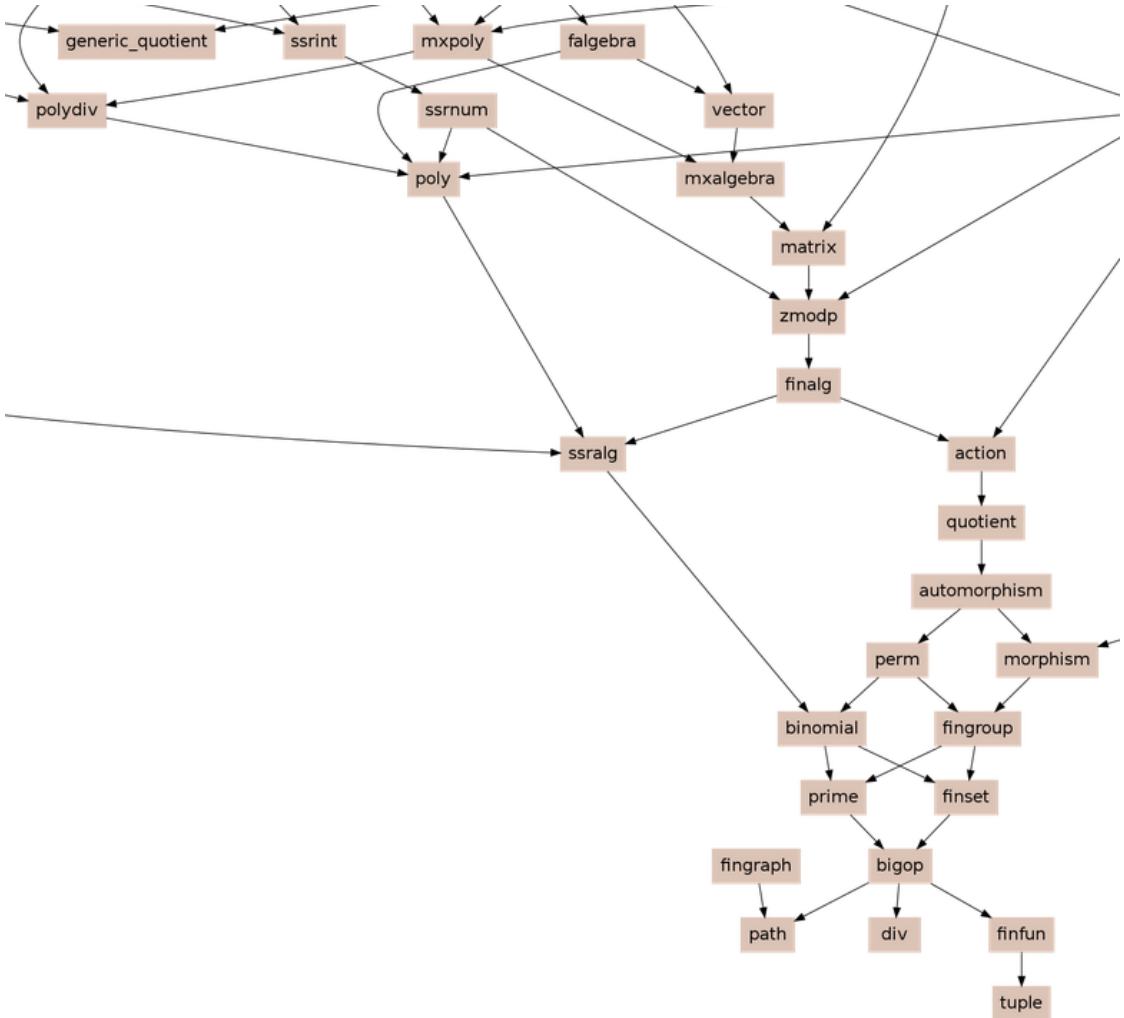
参考ファイル➡fintype_example.v

MathComp ライブラリ



図は http://ssr2.msr-inria.inria.fr/doc/ssreflect-1.4/ より

MathComp ライブラリ (拡大)



図は http://ssr2.msr-inria.inria.fr/doc/ssreflect-1.4/ より

tuple.v: Fixed-size リスト

- 依存型の代表的な例
  - 帰納的に定義される型としてよく定義する:

```
Inductive vec (A : Set) : nat -> Set :=
| vnil : vec A 0
| vcons : A -> forall n : nat, vec A n -> vec A (S n).
```

⇒ 扱いにくい, リストに既にある定義や補題等の再開発は必要

- SSReflect では, リストのライブラリを再利用

```
Structure tuple_of (n : nat) (T : Type) : Type :=
  Tuple {tval :> seq T; _ : size tval == n}.
```

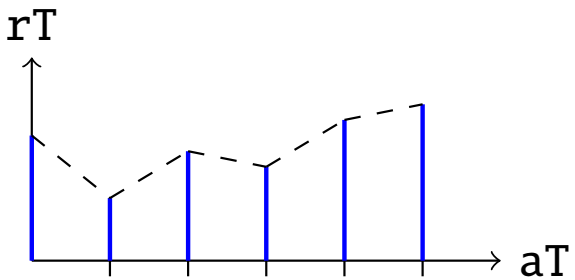
- 記号:
  - 型: `n.-tuple T`
  - 値: `[tuple of s]`
    - 例えば: `[tuple of [seq x * 2 | x <- [:: 1; 2; 3]]]`

参考ファイル➡tuple_example.v

finfun.v: グラフとしての関数

- Coq は intensional である
  - 同じ入力/出力関係があっても, アルゴリズムが違ったら, 二つの関数は等しくない (公理として追加は可能 [CDT])
- finfun.v は extensional な関数の型を提供する:

```
Variables (aT : finType) (rT : Type).
Inductive finfun_type :=
  Finfun of #|aT|.-tuple rT.
```



(NB: fintype.v と tuple.v を利用)

- 記号:
  - 型: {ffun aT ->rT}
  - 値: (g : aT ->rT) なら, [ffun x =>g x]
- 外延性を補題として回復:

```
Lemma ffunP : forall (f1 f2 : {ffun aT -> rT}), f1 =1 f2 <-> f1 = f2
Lemma ffunE : forall(g : aT -> rT), [ffun x => g x] =1 g
```

finset.v: 有限集合

- T 型をもつ要素の有限集合は#|T|長いビットマスクとして定義:

```
Inductive set_type (T : finType) := FinSet of {ffun pred T}.
```

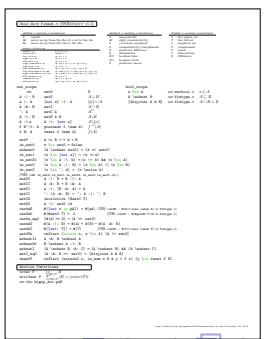
- 型: {set T}; 値: [set x | P]
- s : {set T}なら, t \in s は書けます
  - SetDef.pred_of_set を用いて, set_type を predType として登録
- 有限集合の定義の例:

```
Definition set0 := [set x : T | false].
Definition setU A B := [set x | (x \in A) || (x \in B)].
```

- 外延性: Lemma setP A B : A =i B <-> A = B.

参考資料: ssrbool_doc.pdf

- 記号 (一部, ssrbool.v, fintype.v), 補題:



bigop.v

- 数学と計算機科学 [GKP94, Chapter 2] にな不可欠ライブラリ
  - 奇数位数定理の成功に大事なライブラリ [BGBP08]
- iterated operations: $+ \to \sum$, $\times \to \prod$, $\cup \to \bigcup$, $\cap \to \bigcap$ 等
- 記号:

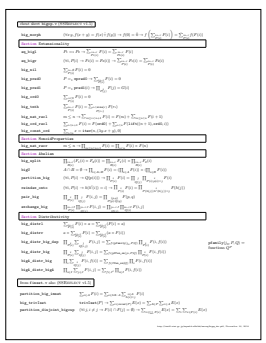
紙上	SSReflect
bigop.v	
$\sum_{\substack{0 \le i < n \\ P(i)}} F(i)$	`\big[addn/0]_(0 <= i < n | P i) F i` `\sum_(0 <= i < n | P i) F i`
$\prod_{\substack{0 \le i < n \\ P(i)}} F(i)$	`\big[muln/1]_(0 <= i < n | P i) F i` `\prod_(0 <= i < n | P i) F i`
finset.v	
$\bigcup_{\substack{i \\ P(i)}} F(i)$	`\bigcup_(i | P i) F i`
$\bigcap_{\substack{i \\ P(i)}} F(i)$	`\bigcap_(i | P i) F i`

- index は `finType` 型である
  - `'I_n(Ordinal)`, `{ffun aT -> rT}` (`finfun.v`) 等

bigop.v: 補題の例

ゴール (前)	タクティック	ゴール (後)
`======================` `...` `\sum_(0 <= i < n.+2) i` `...`	`rewrite big_nat_recr //=.`	`======================` `...` `\sum_(0 <= i < n.+1) i` `+ n.+1` `...`

参考資料: bigop_doc.pdf



bigop.v: 練習

- ```
  Lemma gauss : forall n : nat, 2 * (\sum_(0 <= x < n.+1) x) = n * n.+1.
  Proof. ...
  ```

  ヒント: big_nat_recr, big_nil, ssrnat.v で十分

- ```
  Lemma bigop_test : (a + b)^2 = a^2 + 2 * a * b + b^2.
  Proof. ...
  ```

  ただし, bigA_distr_big を使う

  参考ファイル➡bigop_example.v

- 他の応用例: スライド 108

fingroup.v: 群論

- 群は有限集合を用いて作る [GMR+07, BMR+, Tas14]
- finGroupType は group laws を持つ型である
  - 例えば, 置換群 S_3:

$\cdot \times \cdot$	(01)	(02)	(12)	(012)	(021)
(01)	1	(021)	(012)	(12)	(02)
(02)	(012)	1	(021)	(01)	(12)
(12)	(021)	(012)	1	(02)	(01)
(012)	(02)	(12)	(01)	(021)	1
(021)	(12)	(01)	(02)	1	(021)

	$.^1$
(01)	(01)
(02)	(02)
(12)	(12)
(012)	(021)
(021)	(012)

- gT : finGroupType なら, 集合 A : {set gT}は次の命題を満たすと, 群である:

```
Definition group_set A := (1 \in A) && (A * A \subset A).
```

- 生成集合 (記号: << A >>):

```
Definition generated A := \bigcap_(G : groupT | A \subset G) G.
```

fingroup.v: 正規化群

- 定義:
  - 共軛作用 (記号: x1 ^ x2):

```
Definition conjg (T : finGroupType) (x y : T) := y^-1 * (x * y).
```

(NB: 記号: 「A :^ x」は「conjg^~ x @: A」)
  - 正規化群 (記号: 'N(A)):

```
Definition normaliser A := [set x | A :^ x \subset A].
```

  - 正規関係 (記号: A <| B):

```
Definition normal A B := (A \subset B) && (B \subset 'N(A)).
```

- 練習 (perm.v):
  - 置換群'S_3 は可換群でないことを示す
  - {(012), (021), 1} から生成された A_3 群は正規であると示す
  - 参考ファイル ➡permutation_example.v

matrix.v

線型代数, 行列の定義とその関数と定理

- 型 R の要素を持つ $m \times n$ の行列は'I_m *'I_n から R までの finfun 関数として定義されている:

```
Variable (R : Type) (m n : nat).
Inductive matrix := Matrix of {ffun 'I_m * 'I_n -> R}.
Notation "''M[' R ]_ ( m , n )" := (matrix R m n).
```

- M : 'M[R]_(m, n) があれば, M m0 n0 は (m_0, n_0) 番目の要素となる
- ssralg.v で環が ringType 型として定義されている [GGMR09]
- R は ringType であれば, 行列の「+」等を ssralg.v から取得する
- 行列の定義の例:

```
Definition odd_bool : 'M[bool]_(m, n) := \matrix_(i < m, j < n) odd (i + j).
Definition odd_R : 'M[R]_(m, n) := \matrix_(i < m, j < n) (odd (i + j))%:R.
```

応用例: スライド 110

Outline

定理証明支援系とは
定理証明支援系の概要
定理証明支援系の歴史
定理証明支援系の応用例
数学の証明の形式化
ソフトウェアの形式検証
定理証明支援系 Coq/SSReflect の入門
Coq による形式証明の原理
形式モデルと仕様の作成
Coq と SSReflect の関係
形式証明の基本
SSReflect ライブラリの紹介
SSReflect ライブラリの概要
基礎ライブラリ
総和と総乗
群と代数
関連するトピックス
命令型プログラムの検証
アセンブリ言語への応用
C 言語への応用
数学の証明の形式化
情報理論の形式化
符号理論の形式化

低レベルプログラムの形式検証の基本

- コンピュータのメモリを連想リストとして形式化する [MAY06]
  - D は順序集合 (domain), CD は決定可能な同値関係を持つ型 (codomain):

```
Definition l := D.A.
Definition ltl := D.ltA.
Definition v := CD.A.
Inductive map : Type :=
  mk_map (s : seq (l * v)) of ordered D.ltA (unzip1 s).
```

    ⇒ 依存型の応用例
- 整数型の形式化 [AM08]
  - 文字型: `int 8`, ポインタ: `int ptr_len` 等
  - `tuple.v` と同じ
- 仕様の shallow embedding:
  - 仕様言語として, `Prop` のものを使う:

```
Definition assert : Type := store -> heap -> Prop.
```

- プログラムの構文は帰納的に定義する (deep embedding)

ホーア論理による検証

- ホーアトリプル: P を満たす状態からプログラム c を実行すると, 停止の際, 状態が Q を満たす:

$$\{P\}c\{Q\}$$

  P, Q は Coq の論理でもよい
- 形式体系 (次のスライドに参照):
  - ホーアトリプルの推論規則
    - 1 つのプログラミング言語の要素 $\leftrightarrow$ 1 つの推論規則
  - 健全と “完全” である
    - 意味操作論と合意し, 証明できないものはない
- 形式検証:
  - 理論上:
    - ループ不変条件があれば, 事前/事後条件から全部の中間条件を計算できる
    - 基本的な命令の正しさの証明を自動定理証明器に任せる
  - 実際上:
    - ループ変数条件は簡単に見つからない
    - 現在の自動化は不十分
    - 検証を考えず実装されたプログラムの検証より, 検証に相応しいプログラムを作り直したほうがよい

  $\Rightarrow$ 対話的な検証ができるように努める

ホーア論理の形式化

ホーアトリプルの形式定義 (パラメーター hoare0 を持つモジュール):

```
Inductive hoare : assert -> cmd -> assert -> Prop :=
| hoare_hoare0 : forall P Q c,
  hoare0 P c Q -> {{ P }} c {{ Q }}
```

```
| hoare_seq : forall P Q R c d,
  {{ P }} c {{ Q }} -> {{ Q }} d {{ R }} ->
  {{ P }} c ; d {{ R }}
```

$$\frac{\{P\}c\{Q\} \quad \{Q\}d\{R\}}{\{P\}c;d\{R\}}$$

```
| hoare_conseq : forall P P' Q Q' c,
  P ===> P' -> {{ P' }} c {{ Q' }} -> Q' ===> Q ->
  {{ P }} c {{ Q }}
```

$$\frac{P \to P' \quad \{P'\}c\{Q'\} \quad Q' \to Q}{\{P\}c\{Q\}}$$

```
| hoare_while : forall P t c,
  {{ fun s h => P s h /\ eval t (s, h) }} c {{ P }} ->
  {{ P }} while t c {{ fun s h => P s h /\ ~ eval t (s, h) }}
```

$$\frac{\{\lambda s, h.P\,s\,h \wedge \mathtt{eval}(t,(s,h))\}c\{P\}}{\begin{array}{c}\{P\}\\ \mathtt{while}(t)\{c\}\\ \{\lambda s, h.P\,s\,h \wedge \neg\mathtt{eval}(b,(s,h))\}\end{array}}$$

```
| hoare_ifte : forall P Q t c d,
  {{ fun s h => P s h /\ eval t (s, h) }} c {{ Q }} ->
  {{ fun s h => P s h /\ ~ eval t (s, h) }} d {{ Q }} ->
  {{ P }} ifte t c d {{ Q }}.
```

$$\frac{\begin{array}{c}\{\lambda s, h.P\,s\,h \wedge \mathtt{eval}(t,(s,h))\}c\{Q\}\\ \{\lambda s, h.P\,s\,h \wedge \neg\mathtt{eval}(t,(s,h))\}d\{Q\}\end{array}}{\{P\}\mathtt{if}\,t\,\mathtt{then}\,c\,\mathtt{else}\,d\{Q\}}$$

分離論理 [Rey02]

- 通常のホーア論理でポインタは扱いにくい
- ポインタを用いたホーア論理の拡張
  - 論理に $\mapsto, \star, \mathrel{-\!\!\star}$ の追加
- ポインタの扱いの形式検証は簡単になる (aliasing の性質は自明になる, 二つの論理積 $\wedge, \star$ は自然である)
- 例えば, メモリアップデートのホーアトリプル:

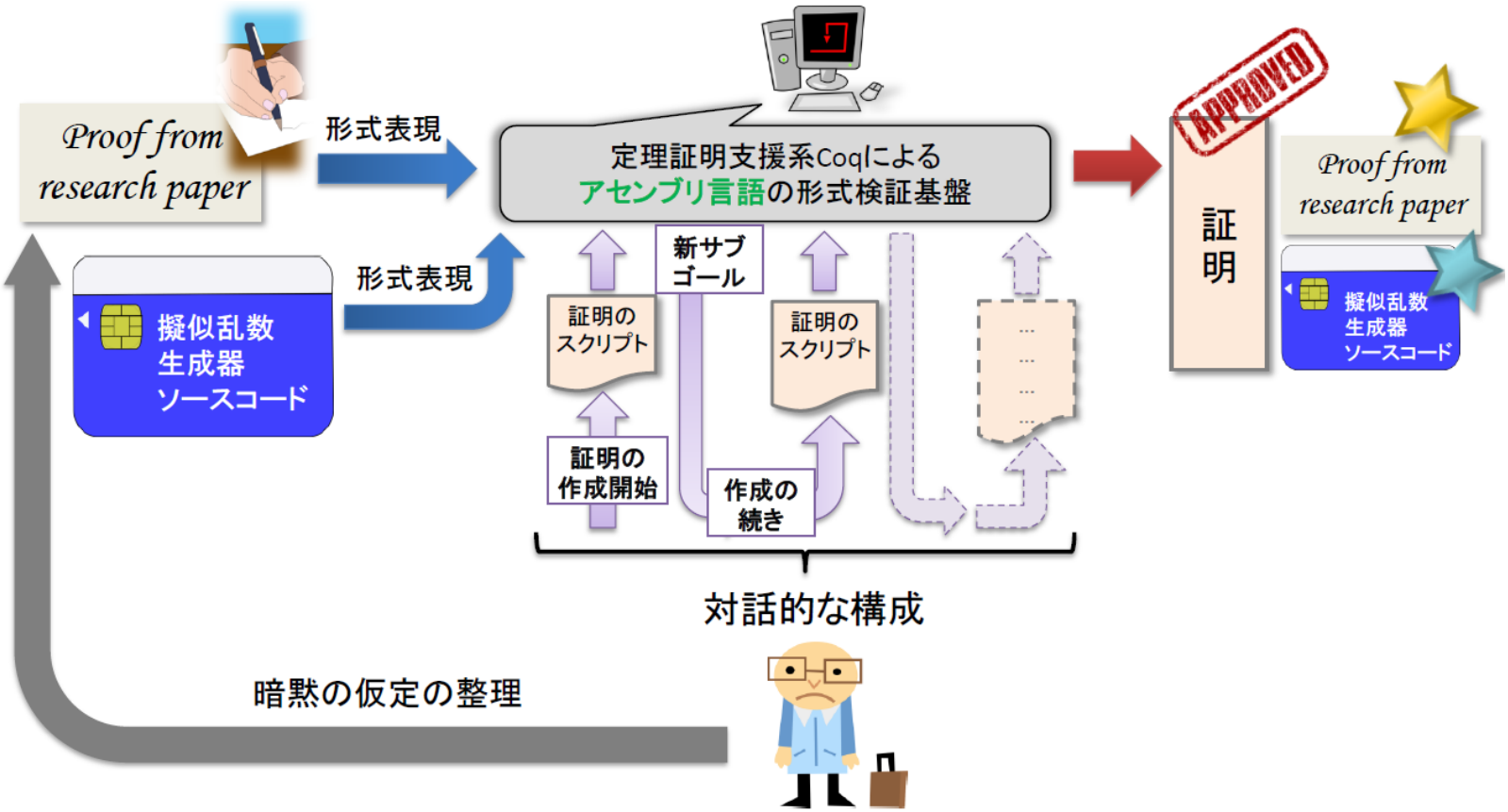
$$\{\exists e''.(e \mapsto e'' \star (e \mapsto e' \mathrel{-\!\!\star} Q))\}[e] := e'\{Q\}$$

- 分離論理によるメモリアップデート:

$$\dfrac{\dfrac{\dfrac{(e \mapsto e_0) \to (E \mapsto e_0) \qquad \dfrac{\overbrace{P \star (e \mapsto e')}^{\text{アップデートの後の条件}} \to Q}{P \to (e \mapsto e' \mathrel{-\!\!\star} Q)}\ \text{Adjunction}}{(e \mapsto e_0) \star P \to e \mapsto e_0 \star (e \mapsto e' \mathrel{-\!\!\star} Q)}\ \text{Monotony}}{\underbrace{(e \mapsto e_0)}_{\text{アップデートの前の条件}} \star P \to \underbrace{\exists e''.e \mapsto e'' \star (e \mapsto e' \mathrel{-\!\!\star} Q)}_{\text{メモリアップデートの事前条件}}}\ \text{instantiation}$$

アセンブリプログラムの形式検証実験

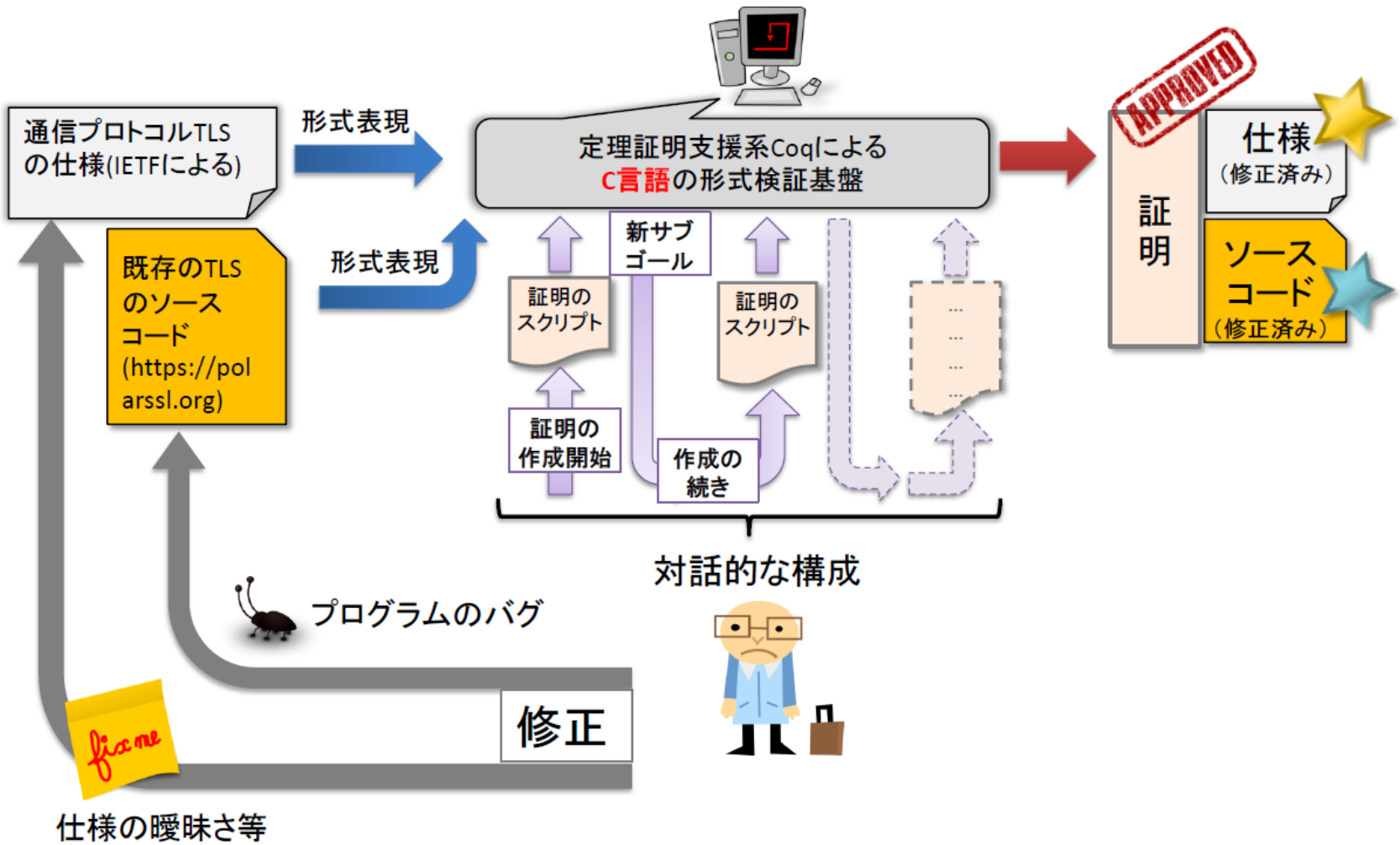
BBS の実装の 暗号学的 安全性の形式検証 [ANY12]



詳細化によってアセンブリで実装された算術関数 [Aff13a]

C 言語のプログラムの形式検証実験

C 言語で実装されたネットワークパケット処理 [AM13, AS14]



分離論理による形式検証

実験の量的な纏め

- アセンブリ (SmartMIPS) プログラム
  - 擬似乱数生成器 BBS [ANY12]
    - ソースコード: 237 行
    - 証明: 正しさ: 4483 行 (Coq); 停止性, 紙上と Coq の証明の間のリンク等: 2374 行
    - while 文, if 文 → goto 文のコンパイラ: 2943 行 (Coq)

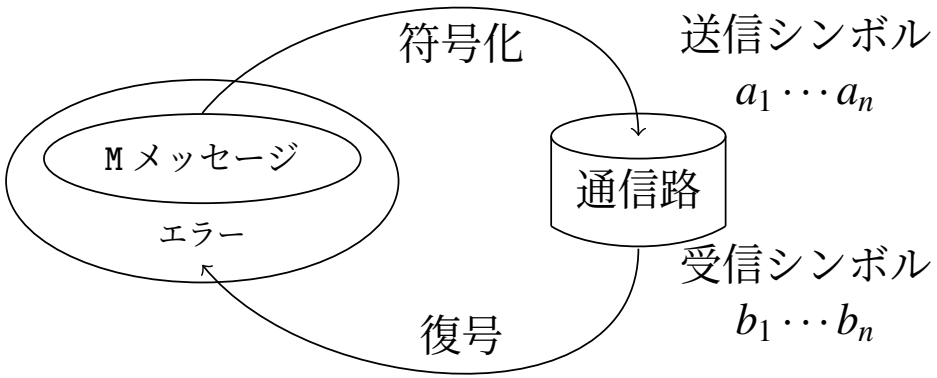
    ⇒ GDB のバグの発見
  - Binary Extended Euclid アルゴリズム [Aff13a]
    - ライブラリソースコード: 313 行 (関数: 25 個)
    - Binary Extended Euclid アルゴリズム: 68 行 (疑似コード: 49 行)
    - 証明: 正しさ: 7746 行; アセンブリと疑似コードの間の simulation: 4753 行
    - Binary Extended Euclid アルゴリズムの simulation の証明: 1466 行
- C プログラム [AS14]
  - PolarSSL の parsing 関数
    - ソースコード: 161 行 (コメント, debugging 情報を含む); Coq モデル: 132 行 (12 行の patch を含む)
    - TLS の RFC: 1258 行 (Coq); 元のテキストファイル: 102 頁 (約 57% を形式化済)
    - 正しさの証明: 4153 行 (Coq) (24 行の Coq↔ 1 行の C)
    - 基盤: 12621 行 (Coq) (簡単な例を含む)

  ⇒ バグの発見 (NB: パケットの長さのチェックは不十分 (3 個, 2014 年 6 月の GnuTLS の SessionID バグに近い, Heartbleed なみのバグ), TLS extensions チェックも不十分と言える, 暗黙の仮定)

情報理論の形式化

符号化システム

- メッセージ: M という集合の要素
- 通信路: 雑音がある
- 符号化: 予め冗長な情報を加える




```
Definition encT :=
  {ffun M -> n.-tuple A}.
```

- 復号: 送信後

```
Definition decT :=
  {ffun n.-tuple B -> option M}.
```

- 符号化システム:

```
Record code := mkCode { enc : encT ; dec : decT }.
Definition CodeRate (c : code) := log (INR #| M |) / INR n.
```

情報理論と bigop.v の関係

情報理論で $\sum, \prod, \bigcap, \bigcup$ をしばしば使う; 例えば:

- 有限集合上の確率分布:
  - 定義域に所属する任意の a に対して 0 以上の実数を与える関
  - 定義域に対する総和が 1 にならなければならない

  $\Rightarrow$ dependent record を使う:

```
Record dist := mkDist {
  pmf :> A -> R ;
  pmf0 : forall a, 0 <= pmf a ;
  pmf1 : \rsum_(a in A) pmf a = 1 }.
```

  コアーション:>のおかげで, (pmf P) は P a で書ける

- 確率: $\Pr_P[E] = \sum_{a \in E} P(a)$:

```
Definition Pr (P : dist A) (E : {set A}) := \rsum_(a in E) P a.
```

- エントロピー:

```
Variable P : dist A.
Definition entropy := - \rsum_(a in A) P a * log (P a).
```

確率分布の簡単な性質

bigop.v を使った例:

- P1 : dist A, P2 : dist B, $f : (a, b) \mapsto P_1(a)\,P_2(b)$
  - f は確率分布?
  - $\sum_{ab \in A \times B} f(ab) = 1$? (ゴール)
  - $\sum_{a \in A} \sum_{b \in B} P_1(a)\,P_2(a) = 1$? (pair_big 補題)
  - $\sum_{a \in A} \sum_{b \in B} P_1(a)\,P_2(a) = \sum_{a \in A} P_1(a)$? (確率分布の定義)
  - $\sum_{b \in B} P_1(a)\,P_2(a) = P_1(a)$? (eq_bigr 補題)
  - $P_1(a) \sum_{b \in B} P_2(a) = P_1(a)$? (big_distrr 補題)
  - $P_1(a) \cdot 1 = P_1(a)$? (確率分布の定義)
- P : dist A, $f : A^n \to \mathbb{R}; t \mapsto \prod_{i<n} P(t_i)$ (NB: A^n は n.-tuple A の略)
  - f は確率分布?
  - $\sum_{t \in A^n} f(t) = 1$? (ゴール)
  - $\sum_{g \in A^{[1,n]}} \prod_{i<n} P(g(i)) = 1$? (reindex_onto 補題)
  - $\prod_{i<n} \sum_{a \in A} P(a) = 1$? (bigA_distr_bigA 補題)
  - $\prod_{i<n} \sum_{a \in A} P(a) = \prod_{i<n} 1$? (big_const_ord 補題)
  - $\sum_{a \in A} P(a) = 1$? (eq_bigr 補題 + 確率分布の定義)

通信路符号化定理

- 通信路: 確率遷移行列

```
Notation "'`Ch_1'" :=
  (A -> dist B).
```

$$\begin{pmatrix} W(b_1|a_1) & W(b_2|a_1) & \cdots & W(b_{|B|}|a_1) \\ W(b_1|a_2) & W(b_2|a_2) & \cdots & W(b_{|B|}|a_2) \\ \vdots & \vdots & \ddots & \vdots \\ W(b_1|a_{|A|}) & W(b_2|a_{|A|}) & \cdots & W(b_{|B|}|a_{|A|}) \end{pmatrix}$$

- 通信路容量: 相互情報量の上界の最小値として定義 (定義は [AH12, AHS14] に参考) (日本語の概要: [Aff13b])
- 誤り率: 復号の失敗の確率の平均
- 通信路符号化定理の (順) 定理: 通信路 W の通信路容量を cap とする. 符号化率 r に対して, r < cap ならば, 任意に小さい誤り率 epsilon を達成する符号化システム c が存在する:

```
Theorem channel_coding (r : CodeRateType) : r < cap ->
  forall epsilon, 0 < epsilon ->
    exists n M (c : code A B M n), CodeRate c = r /\ echa(W, c) < epsilon.
```

- 通信路符号化定理の逆定理の形式化について [AHS14] に参考
- 量的な纏め: シャノン定理: 8378 行; 標準ライブラリの拡張: 4060 行

符号理論

- 通信路符号化定理によって, 符号が存在すると分かる
- Systematic form の線型符号:
  - チェックシンボル行列:

```
Variable A : 'M['F_2]_(len - dim, dim).
```

  - パリティ検査行列:

```
Definition H : 'M_(len - dim, len) :=
  cast_cols subnKC Hdimlen (row_mx A 1%:M).
```

$$\begin{pmatrix} & 1 & \\ A & & \ddots \\ & & 1 \end{pmatrix}$$

  - 符号化:

```
Definition G : 'M_(dim, len) :=
  cast_cols subnKC Hdimlen (row_mx 1%:M (-A)^T).
```

$$\begin{pmatrix} 1 & & \\ & \ddots & (-A)^T \\ & 1 & \end{pmatrix}$$

線型符号の簡単な性質

ssralg.v と matrix.v を使った例:

```
Lemma G_H_T : G *m H ^T = 0.
```

- $\begin{pmatrix} 1 \parallel (-A)^T \end{pmatrix} \times \begin{pmatrix} A \parallel 1 \end{pmatrix}^T = 0?$ (ゴール)
- $\left(1 \| (-A)^T\right) \times \left(\dfrac{A^T}{1^T}\right)?$ (tr_row_mx 補題)
- $1 \times A^T + (-A)^T * 1^T = 0?$ (mul_row_col 補題)
- $A^T + (-A)^T * 1^T = 0?$ (mul1mx 補題)
- $A^T + (-A)^T * 1 = 0?$ (trmx1 補題)
- $A^T + (-A)^T = 0?$ (mulmx1 補題)
- $(A - A)^T = 0?$ (linearD1 補題)
- $0^T = 0?$ (addrN 補題)
- $0 = 0?$ (trmx0 補題)

結論

- SSReflect によるスクリプトの記述とライブラリデザイン (記号の使い方, 一定の命名規則等) によって, 効率が上る
- 現実的な低レベルプログラムの対話的な形式検証は実用的になりつつある
- Coq/SSReflect と MathComp を用いて, 組合せ論や群論や線型代数などに関する形式検証ができるようになる

[Aff13a] Reynald Affeldt, *On construction of a library of formally verified low-level arithmetic functions*, Innovations in Systems and Software Engineering **9** (2013), no. 2, 59–77.

[Aff13b] ———, 形式的な符号理論に向けて, 数学セミナー **618** (2013), 74–79, 日本評論社.

[Aff14] ———, 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—, 情報処理 **55** (2014), 482–491, 情報処理学会.

[AGN09] Andrea Asperti, Herman Geuvers, and Raja Natarajan, *Social processes, program verification and all that*, Mathematical Structures in Computer Science **19** (2009), no. 5, 877–896.

[AH12] Reynald Affeldt and Manabu Hagiwara, *Formalization of shannon's theorems in ssreflect-coq*, Proceedings of the 3rd International Conference on Interactive Theorem Proving, ITP 2012, Princeton, NJ, USA, August 13-15, 2012, Lecture Notes in Computer Science, vol. 7406, Springer, 2012, pp. 233–249.

[AHS14] Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues, *Formalization of Shannon's theorems*, J. Autom. Reasoning **53** (2014), no. 1, 63–103.

[AM08] Reynald Affeldt and Nicolas Marti, *An approach to formal verification of arithmetic functions in assembly*, Proceedings of the 11th Asian Computing Science Conference on Secure Software and Related Issues, Advances in Computer Science - ASIAN 2006, Tokyo, Japan, December 6-8, 2006, Lecture Notes in Computer Science, vol. 4435, Springer, 2008, pp. 346–360.

[AM13] ———, *Towards formal verification of TLS network packet processing written in C*, Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013, ACM, 2013, pp. 35–46.

[ANY12] Reynald Affeldt, David Nowak, and Kiyoshi Yamada, *Certifying assembly with formal security proofs: The case of BBS*, Sci. Comput. Program. **77** (2012), no. 10-11, 1058–1974.

[AS14] Reynald Affeldt and Kazuhiko Sakaguchi, *An intrinsic encoding of a subset of C and its application to TLS network packet processing*, Journal of Formalized Reasoning **7** (2014), no. 1, 63–104.

[BC04] Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development—Coq'Art: The calculus of inductive constructions*, Springer, 2004.

[BGBP08] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca, *Canonical big operators*, Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2008, Montreal, Canada, August 18-21, 2008, Lecture Notes in Computer Science, vol. 5170, Springer, 2008, pp. 86–101.

[BL09] Sandrine Blazy and Xavier Leroy, *Mechanized semantics for the Clight subset of the C language*, J. Autom. Reasoning **43** (2009), no. 3, 263–288.

[BMR+] Yves Berthot, Assia Mahboubi, Laurence Rideau, Pierre-Yves Strub, Enrico Tassi, and Laurent Théry, *International Spring School on Formalization of Mathematics (MAP 2012), March 12–16, 2012, Sophia Antipolis, France*, Available at: `http://www-sop.inria.fr/manifestations/MapSpringSchool`. Last access: 2014/08/05.

[Bol10] Dominique Bolignano, *Applying formal methods in the large*, Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013, Rennes, France, July 22–26, 2013, Lecture Notes in Computer Science, vol. 7998, Springer, 2010, p. 1.

[Bou97] Samuel Boutin, *Using reflection to build efficient and certified decision procedures*, Proceedings of the 3rd International Symposium on Theoretical Aspects of Computer Software, TACS 1997, Sendai, Japan, September 23–26, 1997, Lecture Notes in Computer Science, vol. 1281, Springer, 1997, pp. 515–529.

[CDT] The Coq Development Team, *The Coq proof assistant—frequenly asked questions*, `http://coq.inria.fr/faq`, Available at: `http://coq.inria.fr/faq`. Last access: 2014/08/26.

[CDT12] ———, *The Coq proof assistant reference manual*, INRIA, 2012, Version 8.4pl3.

[CH86] Thierry Coquand and Gérard Huet, *Concepts mathématiques et informatiques formalisés dans le calcul des constructions*, Tech. Report 515, INRIA Rocquencourt, Apr. 1986.

[CH88] ———, *The calculus of constructions*, Information and Computation **76** (1988), 95–120.

[Chu40] Alonzo Church, *A formulation of the simple theory of types*, The Journal of Symbolic Logic **5** (1940), no. 2, 56–68.

[CN08] Boutheina Chetali and Quang-Huy Nguyen, *Industrial use of formal methods for a high-level security evaluation*, Proceedings of the 15th International Symposium on Formal Methods, FM 2008, Turku, Finland, May 26–30, 2008, Lecture Notes in Computer Science, vol. 5014, Springer, 2008, pp. 198–213.

[dB03] N.G. de Bruijn, *Memories of the AUTOMATH project*, Invited lecture at the Mathematics Knowledge Management Symposium 25–29 November 2003 Heriot-Watt University, Edinburgh, Scotland, Nov. 2003, Available at: `http://www.win.tue.nl/automath/aboutautomath.htm` Last access: 2014/08/21.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry, *A machine-checked proof of the odd order theorem*, Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013, Rennes, France, July 22–26, 2013, Lecture Notes in Computer Science, vol. 7998, Springer, 2013, pp. 163–179.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau, *Packaging mathematical structures*, Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, Munich, Germany, August 17-20, 2009, Lecture Notes in Computer Science, vol. 5674, Springer, 2009, pp. 327–342.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete mathematics: A foundation for computer science*, Addison-Wesley, 1994.

[GM10] Georges Gonthier and Assia Mahboubi, *An introduction to small scale reflection in coq*, Journal of Formalized Reasoning **3** (2010), no. 2, 95–152.

[GMR+07] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry, *A modular formalisation of finite group theory*, Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Lecture Notes in Computer Science, vol. 4732, Springer, 2007, pp. 86–101.

[GMT08] Georges Gonthier, Assia Mahboubi, and Enrico Tassi, *A small scale reflection extension for the Coq system*, Tech. Report RR-6455, INRIA, 2008, Version 14 (March 2014).

[Gon05] Georges Gonthier, *A computer-checked proof of the four colour theorem*, Tech. report, Microsoft Research, Cambridge, 2005, Available at: `http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf`. Last access: 2014/08/04.

[Gon08] ———, *Formal proof—the four-color theorem*, Notices of the American Mathematical Society **55** (2008), no. 11, 1382–1393.

[GT12] Georges Gonthier and Enrico Tassi, *A language of patterns for subterm selection*, Proceedings of the 3rd International Conference on Interactive Theorem Proving, ITP 2012, Princeton, NJ, USA, August 13-15, 2012, Lecture Notes in Computer Science, vol. 7406, Springer, 2012, pp. 361–376.

[Hal08] Thomas C. Hales, *Formal proof*, Notices of the American Mathematical Society **55** (2008), no. 11, 1370–1380.

[Hal12] ———, *Lessons learned from the flyspeck project*, International Spring School on Formalization of Mathematics (MAP 2012), March 12–16, 2012, Sophia Antipolis, France, Mar. 2012, Available at: `http://www-sop.inria.fr/manifestations/MapSpringSchool/contents/ThomasHales.pdf` Last access: 2014/08/22.

[Hal14] ———, *Solovyev's formal computations in HOL light*, Workshop on Formalization of Mathematics in Proof Assistants, Institut Henri Poincaré, May 2014, Oral presentation.

[Har06] John Harrison, *Towards self-verification of HOL Light*, Proceedings of the 3rd International Joint Conference on Automated Reasoning, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Lecture Notes in Computer Science, vol. 4130, Springer, 2006, pp. 177–191.

[How80] William A. Howard, *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, ch. The formulae-as-types notion of construction, pp. 479–490, Academic Press Inc., Sep. 1980, Original paper manuscript from 1969.

[Kle10] Gerwin Klein, *From a verified kernel towards verified systems*, Proceedings of the 8th Asian Symposium on Programming Languages and Systems, APLAS 2010, Shanghai, China, November 28–December 1, 2010, Lecture Notes in Computer Science, vol. 6461, Springer, 2010, pp. 21–33.

[Ler09] Xavier Leroy, *A formally verified compiler back-end*, J. Autom. Reasoning **43** (2009), no. 4, 363–446.

[Mah14] Assia Mahboubi, *Computer-checked mathematics: a formal proof of the odd order theorem*, Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014, Vienna, Austria, July 14–18, 2014, ACM, Jul. 2014, Article no. 4.

[MAY06] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa, *Formal verification of the heap manager of an operating system using separation logic*, Proceedings of the 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Lecture Notes in Computer Science, vol. 4260, Springer, 2006, pp. 400–419.

[MT13] Assia Mahboubi and Enrico Tassi, *Canonical structures for the working Coq user*, Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013, Rennes, France, July 22–26, 2013, Lecture Notes in Computer Science, vol. 7998, Springer, 2013, pp. 19–34.

[Nip14] Tobias Nipkow, *Tame graph enumeration in Flyspeck*, Workshop on Formalization of Mathematics in Proof Assistants, Institut Henri Poincaré, May 2014, Oral presentation.

[PM92] Christine Paulin-Mohring, *Inductive definitions in the sytem coq rules and properties*, Tech. Report 92–49, LIP, École Normales Supérieure de Lyon, Dec. 1992.

[Pot03] Loïc Pottier, *Preuves formelles en Coq*, Notes de cours du DEA de mathématiques, Université de Nice-Sophia Antipolis, Jan. 2003, Available at: `http://www-sop.inria.fr/lemme/Loic.Pottier/coursDEA2003.pdf`. Last access: 2014/08/23.

[PW14] Álvaro Pelayo and Michael A. Warren, *Homotopy type theory and Voevodsky's univalent foundations*, Bull. Americ. Math. Soc. **51** (2014), no. 4, 597–648, Article electronically published on May 9, 2014.

[Rey02] John C. Reynolds, *Separation logic: A logic for shared mutable data structures*, Proceedings of the17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, IEEE Computer Society, 2002, pp. 55–74.

[Rus08] Bertrand Russell, *Mathematical logic as based on the theory of types*, American Journal of Mathematics **30** (1908), no. 3, 222–262.

[ST14] Matthieu Sozeau and Nicolas Tabareau, *Universe polymorphism in Coq*, Proceedings of the 5th International Conference on Interactive Theorem Proving, Vienna, Austria, July 14-17, 2014, Lecture Notes in Computer Science, vol. 8668, Springer, 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, pp. 499–514.

[Tas14] Enrico Tassi, *Mathematical components, a large library of formalized mathematics*, Workshop on Formalization of Mathematics in Proof Assistants, Institut Henri Poincaré, May 2014, Oral presentation.

[vH02] Jean van Heijenoort, *From Frege to Gödel—a source book in mathematical logic, 1879-1931*, Harvard University Press, 2002.

[Voe14] Vladimir Voevodsky, *Univalent foundations*, March 2014, Lecture at IAS. Available at: `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2014_IAS.pdf`. Last access: 2014/07/30.

[Wie14] Freek Wiedijk, *Formal proof done right*, Workshop on Formalization of Mathematics in Proof Assistants, Institut Henri Poincaré, May 2014, Oral presentation.

[WKS+09] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish, *Mind the gap*, Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009, Munich, Germany, August 17–20, 2009, Lecture Notes in Computer Science, vol. 5674, Springer, 2009, pp. 500–515.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr, *Finding and understanding bugs in C compilers*, Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, ACM, 2011, pp. 283–294.