定理証明支援系 Coq による形式検証

集中講義@京都大学大学院理学研究科数学・数理解析専攻数理解析系

2015/7/24 版

アフェルト レナルド

産業技術総合研究所

2015 年 7 月 21(火)–24 日 (金)

本講義

概要

- 内容: Coq/SSReflect/MathComp 入門
  - 最初に, 型理論の実装の一つである Coq を説明する. その次に, Coq の拡張である SSReflect の考え方と具体的な記述方法を説明する. 最後に, ライブラリ MathComp を紹介し, その基本的な使い方を説明する.
  - Coq (フランス国立情報学自動制御研究所)
  - SSReflect, MathComp (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
- 目的: 本講義を受講することによって, 参加者は 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
- 成績評価方法: 次の三つの課題から選択する
  1. Coq/SSReflect を用いた簡単な形式証明を実行/整理/構築しなさい (初心者向け)
  2. MathComp を使った問題の定理を証明しなさい (ある程度の Coq 経験者向け)
  3. 学会や雑誌に発表済みの Coq か SSReflect による数学の形式化を調査し (例: Univalent Foundations, 四色定理, 奇数位数定理等), その内容の基本的な形式定義と言明（定理と主な補題）を紙上の証明と比較し, 形式証明の有効性について考察しなさい; 評価はレポート (数ページ以内) にて行う

本講義 I

内容

- 材料: http://staff.aist.go.jp/reynald.affeldt/ssrcoq
  - 本スライド + group_commented.pdf
  - Coq ファイル
    - 18 枚の blah_example.v デモファイル (参考ファイル ➡blah_example.v) : logic_example.v, ssrnat_example.v, predicative_example.v, dependent_example.v, ssrbool_example.v, tactics_example.v, view_example.v, eqtype_example.v, fintype_example.v, tuple_example.v, implicit_example.v, mybigop_example.v, bigop_example.v, finset_example.v, bigop2_example.v, group_example.v, permutation_example.v, matrix_example.v
    - 練習 exo0-40 を含む (参考ファイル ➡blah_example.v, exo?)
    - HTML ドキュメンテーション (coqdoc による)
    - proviola アニメーション [TGMW10]
  - チートシート: ssrbool_doc.pdf, ssrnat_doc.pdf, bigop_doc.pdf, finset_doc.pdf, fingroup_doc.pdf
- 講師の経験:
  - 分散プログラムの形式化とその応用 [AK02, AKY05, AK08]
  - 低レベルプログラムの形式化とその応用 [MAY06, AM08, MA08]
  - 疑似乱数生成器の実装の暗号学的安全性の形式検証 [ANY12]

本講義 II

内容

  - 詳細化によってアセンブリで実装された算術関数 [Aff13a]
  - シャノン定理の形式化 [AH12, AHS14]
  - C 言語で実装されたネットワークパケット処理 [AM13, AS14]
  - 符号理論の形式化 [Aff13b, AG15]
  - 等
- Coq/SSReflect/MathComp に関する学会発表等の抜粋より
  - 参考文献: スライド 153〜
- [Aff14a] の内容を更新と向上 (元々, [Aff14b] の内容を更新と向上, [Aff14b] は [Aff14c] の内容を詳細化)

Outline

定理証明支援系による形式検証: 動機

- 再確認
  - 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))
    - (物理現象を除いた) ハードウェアへの応用
      (例: マイクロコード) (Intel 社の J. Harrison の研究に参考)
  - 数学の証明の正しさ
    - Kepler 予想の証明の査読 [Hal08] (スライド 16)
    - “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] (スライド 84), seL4 マイクロカーネル [WKS+09] (スライド 88))
  - 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:
  1. 証明の記述を支援（自動化, 記号, 抽象化）
  2. 証明の正しさを保証（型理論による）
- 定理証明支援系の強み:
  - 信頼性が高い:
    カーネル (中核部分) は “小さい” ため (スライド 22), 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, 整礎帰納法を利用できるので, 有限システムに制限されない (モデル検査と比べて)
- 定理証明支援系の例:
  - 型理論に基く: Coq, HOL Light, Isabelle/HOL, Agda 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 等
- 使い方: 対話的に証明を構成する

対話的な証明の流れ (Coq/SSReflect の場合)

ユーザ			定理証明支援系 Coq
言明	`Goal forall n : nat, n + n = 2 * n.`	⇝	型検査
		⇜	ゴール $?_1$
証明の記述 (1/3) (ゴール $?_1$ に対して)	`elim.`	⇝	証明項の構築 (開始)
		⇜	ゴール $?_2$, $?_3$
証明の記述 (2/3) (ゴール $?_2$ に対して)	`rewrite addn0.` `rewrite muln0.` `done.`	⇝	証明項の構築 (続き)
		⇜	ゴール $?_3$
証明の記述 (3/3) (ゴール $?_3$ に対して)	`move=> n IH.` `rewrite addnS.` `rewrite addSn.` `rewrite IH.` `rewrite mulnS.` `rewrite add2n.` `done.`	⇝	証明項の構築 (完了)

参考ファイル ➡ ssrnat_example.v

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

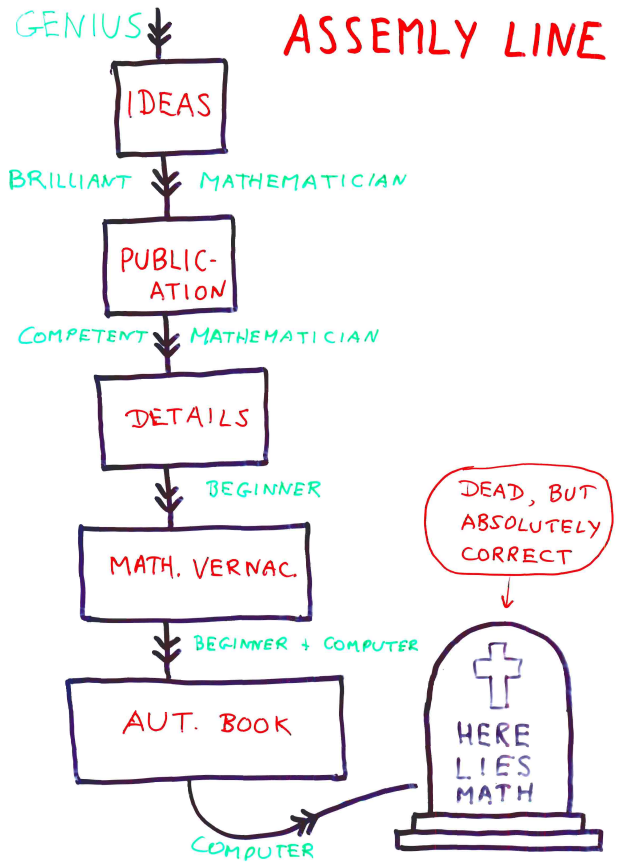
型理論に基づく定理証明支援系の歴史 II

- 1950 年代: カット除去と λ 計算の実行の間 (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 32)
- 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 $\Rightarrow$ 型つきプログラミング言語 ML の発想
- 1984 年: 定理証明支援系 Coq の開発の開始 [CH84]
- 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)

Outline

数学の形式化

[dB03] による:

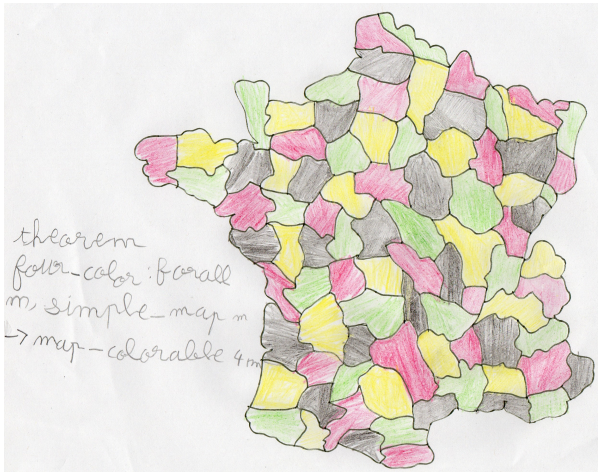


- 目的:
  - ミス, 穴のない証明の作成
  - 証明の向上 (最適化, 短さ)
  - 関連する定理の発見に繋る
- 難しさ:
  - 教科書の 1 頁: 約 1 週間かかる [Hal08, Wie14]
  - 200 頁の教科書: 約 5 人年かかる [Wie14]
- ソフトウェアとハードウェアの検証に繋る
  - 定理のライブラリ
  - 証明記述の技術

四色定理の形式化

歴史

「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
- 誤った証明: A. Kempe, P. G. Tait
- 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
    - 大量の場合分け (10,000 submaps), 一部の計算は IBM 370-168 のアセンブリプログラムに任されていた (1,000,000,000 colorings; 実行は 1200 時間 (約 2ヶ月間) かかった) [Hal13]
    - 実際に, 間もなく, 場合分け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

形式化

- 2000 年ごろ: G. Gonthier と B. Werner (INRIA, Microsoft Research) は Coq で形式化を開始
- 2004 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 (`fourcolor.v` に言明):

```
Theorem four_color (m : map R) : simple_map m -> map_colorable 4 m.
```

  - 証明: 約 60,000 行のスクリプト [Gon05]
- SSReflect(Coq の拡張) の開発のきっかけ; 現在, 数学の形式化以外でも広く利用

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（現在: Airbus 社） Foundation 賞を受賞)
- 2012 年 9 月: 完成; `PFsection14.v` に言明:

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

- 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
  - その他: 再利用性の高い基礎ライブラリ

Kepler 予想の証明の形式化

歴史

「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方格子である」

- 1611 年: J. Kepler が予想を発表
- Hilbert の第 18 問題
- 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 [Hal08]; 実行時間: 約 2,000 時間 (約 3ヶ月間)
  - 2012 年: プログラム ≤10,000 行; 実行時間: 約 20 時間 [Hal12, Hal13]
- 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完全に保証出来ない [Hal08]

Kepler 予想の証明の形式化

形式化プロジェクトの概要

- 2003 年 (の数年後): Flyspeck (Formal Proof of Kepler Conjecture の略) プロジェクト開始
- 形式化に必要な時間の見積は難しい:
  - 2008 年: “Flyspeck may take as many as twenty work-years to complete.” [Hal08]
  - 2012 年: “The Flyspeck project is about 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

Kepler 予想の証明の形式化

形式化の概要

- 一部は Isabelle/HOL: (反例になるかもしれない) グラフの enumeration
  - Hales の「Archive」: 2200 行の Java プログラム → 5128 グラフ, 600 行の ML プログラム → 2771 グラフ (平均サイズ = 13 ノード, 23,000,000 個のグラフの生成と解析, 約 3 時間の計算) [NBS06, Nip14]
  - 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; Kepler 予想の証明に影響なし) [Nip14]
- 主に HOL Light
  - linear programming の結果の形式化
  - non-linear inequalities の検証 (約 500[Hal12]–985 個の数式 [Hal14]) (multivariate polynomials, non-polynomials (arctan, sqrt, etc.); 数千行の C++プログラムは約千行の OCaml に移植) [Hal14]
- HOL Light 用の SSReflect→ 短いスクリプト (2–3 times)
  - 5 つのタクティック (スライド 29 から SSReflect タクティックの詳細な説明)
  - 2 つのライブラリ

⇒ 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ー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

Outline

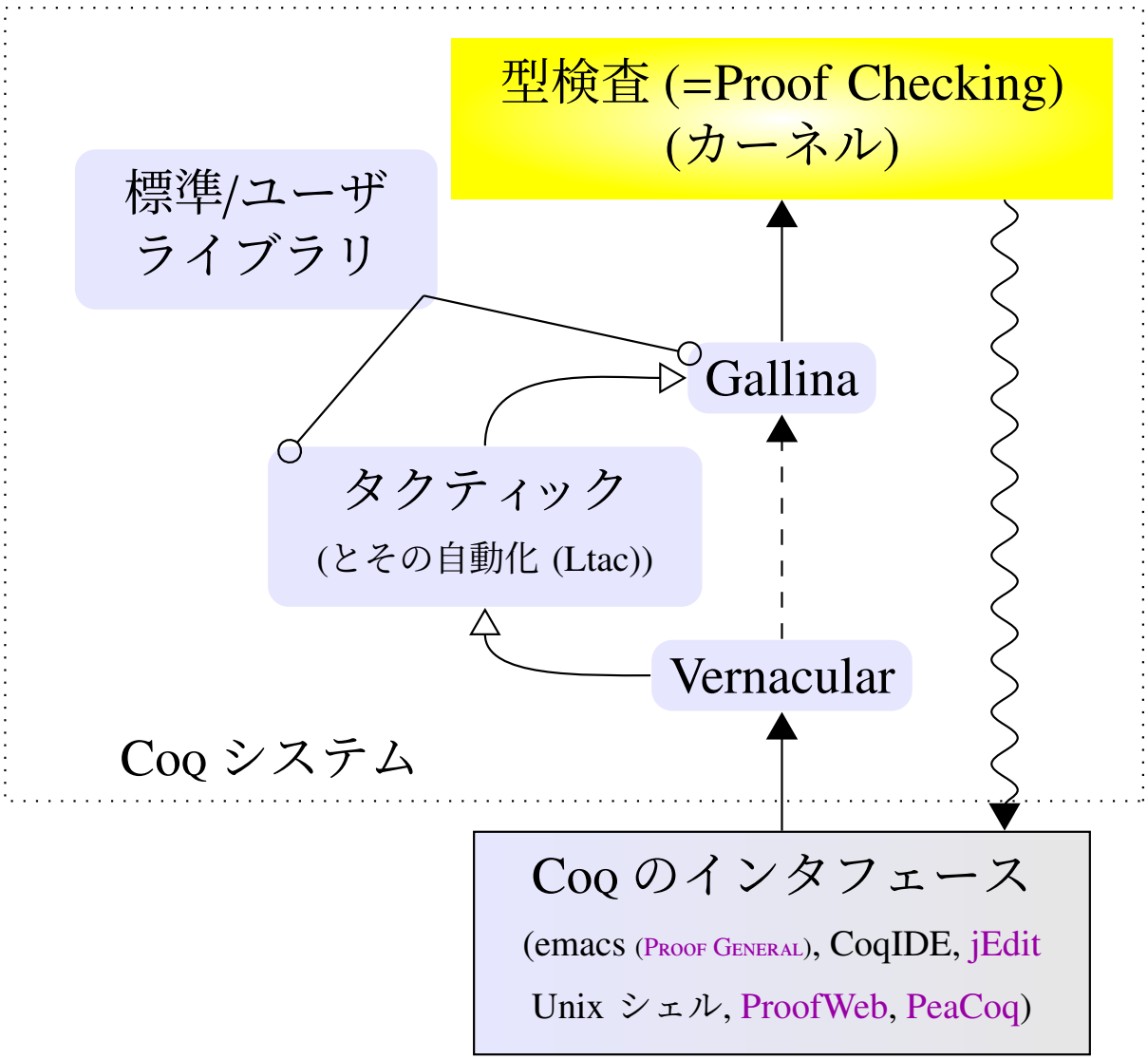
定理証明支援系 Coq

- 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
- 受賞:
  - ACM SIGPLAN Programming Languages Software 2013 賞
  - ACM Software System 2013 賞
- 開発の開始: 1984 年 [CH84]
- 基礎: 型付きプログラミング言語
  - ≃ Calculus of Inductive Constructions [CP90, PM92] (スライド 56 にも参考)
    - Calculus of Constructions [CH84, CH85, CH86, CH88] の拡張

Coq システムの原理

- Propositions-as-types パラダイム:
  - 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 の中核部分とは?
  - ≃ Calculus of Inductive Constructions の項 ($A, B, f, a, \ldots$) + 型検査 (「$\cdot : \cdot$」関係, 決定可能)
  - Coq のカーネルのサイズ（2015/06/17）: 22,494 行 (OCaml) (NB: HOL Light: 約 400 行 [Har06, Hal12], 帰納的（に定義されている）型 (ι-reduction 等) の違い)

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クを提供する

⤳▶: 記述した証明が型検査を通らない場合, ユーザまでフィードバック

Gallina の中核部分

- 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 (中核部分, (NB: スライド 30,スライド 51 にも参考)):

t	:=	Prop	命題のソート
	|	x, A	変数
	|	$A \to B$	非依存型 product
	|	fun x => t	関数抽象
	|	$t_1\ t_2$	関数適用

参考ファイル➡logic_example.v

Gallina の中核部分の型付け規則 (依存型なし)

- Coq のカーネルは次の型 judgment を検査する: $\Gamma \vdash t : A$
  - t = 証明, A = 言明 (Proposition-as-types)
  - Γ = ローカルコンテキスト
    $x_0 : A_0$ (仮定) または $x_1 : A_1 := t_1$(定義) を含む
- [CDT15, Sect. 4.2] による:

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

$$\frac{\Gamma \vdash A \to B : \texttt{Prop} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\, x \Rightarrow t : A \to B}\ \text{Lam} \qquad \text{補題の導入}$$

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

(NB: $A \to B$: Prop? ⇒ スライド 55)

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 (Gallina の中核部分に当る)

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

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

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

Hilbert の公理 S の対話的な証明の流れ (前半)

最初のゴール	$\vdash (A \to B \to C) \to (A \to B) \to A \to C$
	$\rightsquigarrow ?_0$
Lam を適用	$H_1 : A \to B \to C \vdash ?_1 : (A \to B) \to A \to C$
	$?_0 = \lambda H_1 : A \to B \to C.?_1$
	$\rightsquigarrow ?_1$
Lam を 2 回適用	$\Gamma^2 \vdash ?_3 : C$
	$?_1 = \lambda H_2 : A \to B.\lambda H_3 : A.?_3$
	$\rightsquigarrow ?_3$
App(パラメーター B) を適用	$\Gamma \vdash ?_4 : B \to C, \Gamma \vdash ?_5 : B$
	$?_3 = ?_4\ ?_5$
	$\rightsquigarrow ?_4, ?_5$
App(パラメーター A) を適用	$\Gamma \vdash ?_6 : A \to B \to C, \Gamma \vdash ?_7 : A$
	$?_4 = ?_6\ ?_7$
	$\rightsquigarrow ?_6, ?_7$
Var を適用	$?_6 = H_1$
Var を適用	$?_7 = H_3$

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

証明項付きの Hilbert の公理 S の証明

上記のプロセスを続けると, 最終的に次の証明になる:

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

⇒ 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)

Vernacular? スクリプト? タクティック?

- タクティック (後で, 細かく説明する)
  - 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)
- スクリプト $\stackrel{def}{=}$ 連続したタクティック
- Vernacular 構文 (NB: Gallina ではない) (また説明がある):
  - `Lemma`: 証明モードに入る
  - `Qed`: 証明項を検査し, 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

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　参考ファイル➡logic_example.v

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

Gallina の中核部分 + 依存型 product

- 項 (中核部分 + 依存型 product, (NB: スライド 51 にも参考)):

t	:=	`Prop` | `Set` | `Type`	命題のソート
	|	x, A	変数
	|	`forall` $x : A, B$	依存型 product
	|	$A \to B$	非依存型 product
	|	`fun` x => t	関数抽象
	|	$t_1\ t_2$	関数適用

- $A \to B$ が `forall` $x : A, B$ に一般化される (NB: 返り値の型が入力の値に依存する)(依存型)
- `fun` は型 `forall`/$\to$ の値, 適用で消費する
- `Prop`, `Set`, `Type` の違いは後で説明する (スライド 53)

Gallina の中核部分の型付け規則 (依存型なし/ありの違い)

依存型なし

$$\Gamma \vdash A \to B : \left\{ \begin{array}{l} \texttt{Prop} \\ \texttt{Set} \\ \texttt{Type}_i \end{array} \right.$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\, x \Rightarrow t : A \to B}\ \text{Lam}$$

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

依存型あり

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

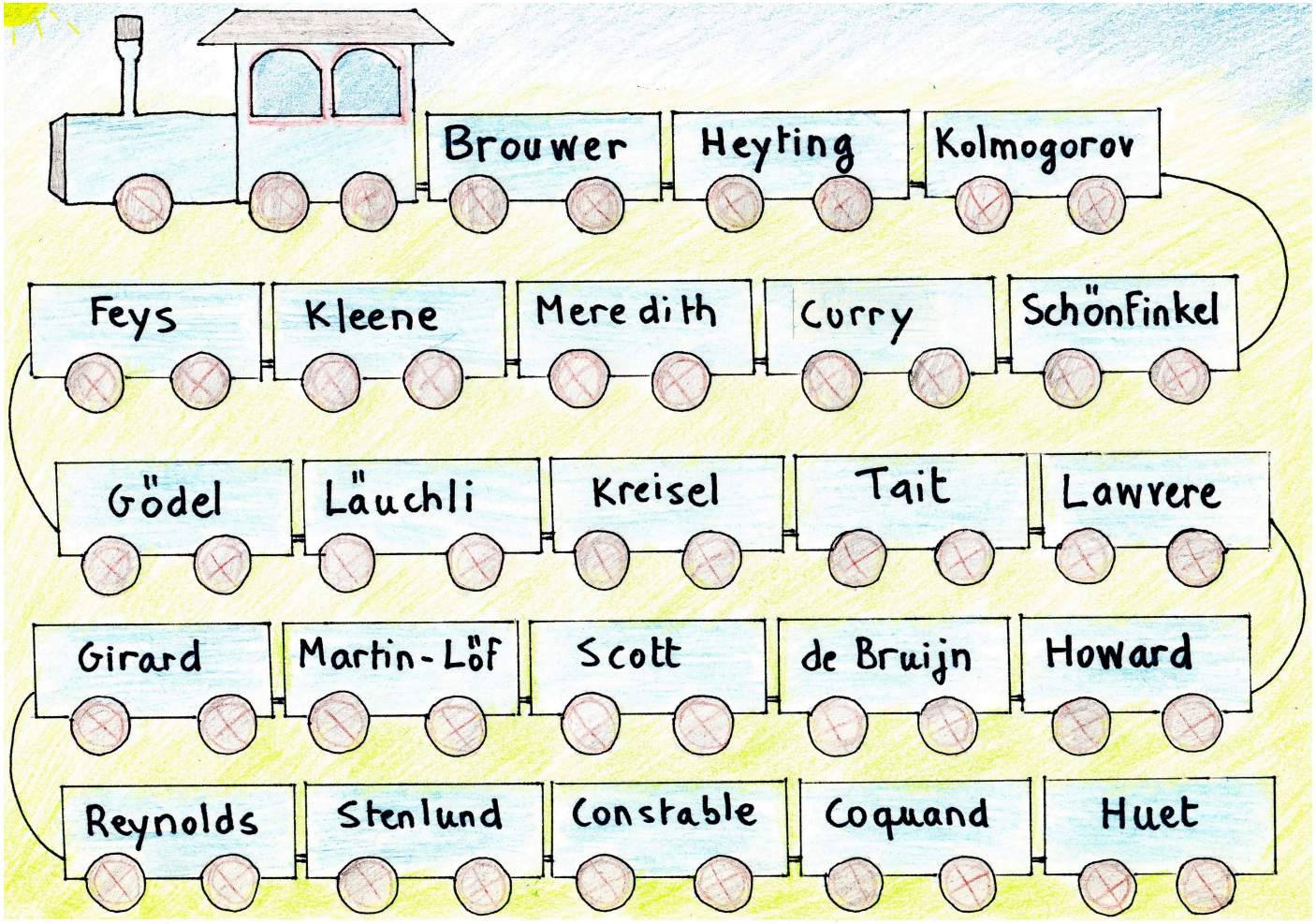
$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \texttt{fun}\, x \Rightarrow t : \texttt{forall}\, x : A, B}\ \text{Lam}$$

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

仮定の導入

補題の利用

Curry-Howard 同型対応 (1/2)



[SU98]

Curry-Howard 同型対応 (2/2)

- 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; ...
- Coq は consistent である（すなわち、証明のない型がある）
  - 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]
  - モデルは重要（`Axiom` の追加を議論するため）であるが、その構築は難しい [MW03, Bar10]
  - 一方、強正規化から、`forall` P : `Prop`, P 型を持つ項はないことを示せる
- Brouwer-Kolmogorov-Heyting の意味論:

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

forall P : Prop, P 型を持つ項はない

[Pot03] による

- 補題： $\emptyset \vdash t$: forall $x : T, U$ を満たす t irred. 項は関数抽象である
  - 証明のサイズに関する帰納法と結論の型付け規則に関する場合分け. Lam 規則しか有り得ないことを確認する.
- 定理： $\emptyset \vdash t$: forall P : Prop, P を満たす irred. 項 t はない
  - Ab absurdo. 上記の補題によって、t は Lam 規則から得た関数抽象である. $t =$ fun P : Prop => u にすれば、P : Prop $\vdash u : P$ も成り立つ. 片付け規則によって場合分けし、当たる型付け規則はないことを確認する.
- 系：$\emptyset \vdash t$: forall P : Prop, P を満たす項 t はない
  - 強正規化を使用 (CC: [GN91, Alt93, Geu95] 等; ECC: [Luo90]; CIC: [PM92, Wer94, Ch. 4])

Outline

Vernacular

- 基本的な補題の記述 (スライド 29 にも参考):

  `Lemma` [補題の名前] : [Gallina による型]. `Proof`. [スクリプト] `Qed`.

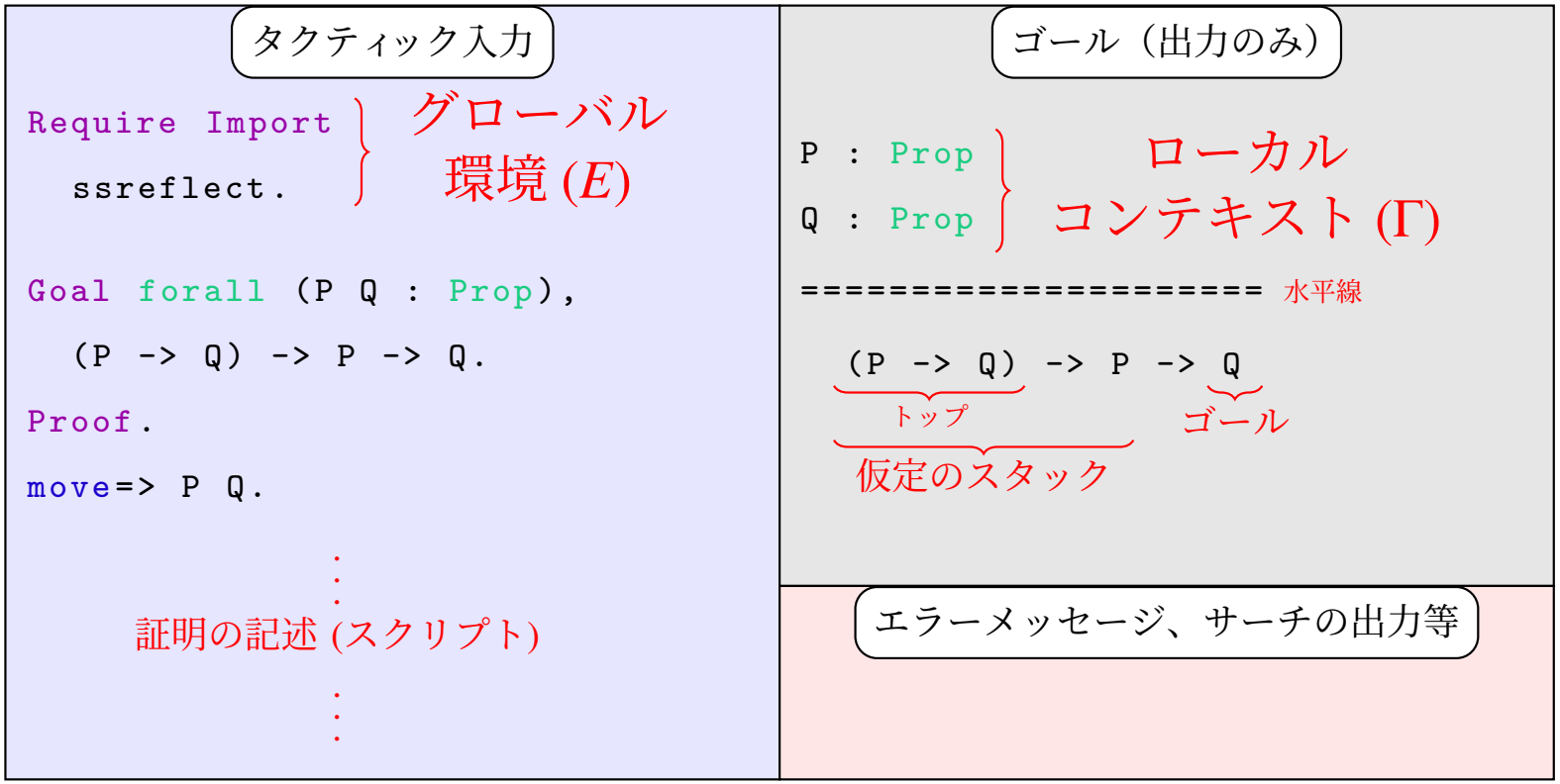
- `Theorem`, `Proposition`, `Corollary`, `Remark`, `Fact` は `Lemma` と同じ
- `Proof`. は見た目のためだけ
- `Qed`. によって, 補題はグローバル環境で登録される
- 登録は不要の時に, `Lemma` の代わりに, `Goal` を使う
- 下記の記述は全部同じ (関連する補題は多いと, `Section` を勧める)

```
Lemma tmp :
 forall P : Prop, P -> P.
Proof. ... Qed.
```

```
Lemma tmp (P : Prop) :
  P -> P.
Proof. ... Qed.
```

```
Section xyz.
Variable P : Prop.
Lemma tmp : P -> P.
Proof. ... Qed.
End xyz.
```

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 または Coq->3 Windows mode layout->hybrid

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 $\stackrel{def}{=}$ 横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```

参考ファイル➡logic_example.v

move=>タクティックによる証明

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

Vernacular の `Show Proof.` で確認できる:

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

- 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 vs. SSReflect

タクティカルと組み合わせると, SSReflect の move は Coq の intro/intros, generalize, clear, pattern 等を一般化する (これから説明する)

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: ゴールは横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, Q : Prop` `PQ : P -> Q` `p : P` `======================` `Q`	`apply: PQ.`	`P, Q : Prop` `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
- 仮定が足りなければ, ユーザにサブゴールが求められる
- 証明項があれば, `exact`/`exact:`も使える

Coq vs. SSReflect

実際に, `apply: H.` は Coq の `refine (H _ ... _).` を一般化 (NB: _の数は自動調整)

参考ファイル➡logic_example.v exo0-3

Outline

論理結合子 True

自然演繹による導入ルール:

$$\frac{}{\Gamma \vdash \mathit{True}}\ \mathrm{True}_i$$

Coq で帰納的に定義された型として定義:

型　ソート

```
Inductive True : Prop :=
  I : True.
```

構成子

構成子の型

True 型の定義によって, 次の導入が行われる:

- constants: True (型), I (True の証明)
- True 型のデータ構造を消費するための elimination ルール: True_rect, True_rec, True_ind (しかし, 使わない)

参考ファイル ➡ logic_example.v

論理結合子 False

型　　ソート

```
Inductive False : Prop := .
```

False 型の定義によって, 次の導入が行われる:

- constants: False だけ
- False 型のデータ構造を消費するための elimination ルール:
  - False_ind : forall P : Prop, False ->P
    - つまり, False の証明があれば, 何でもの P : Prop を証明できる
  - False_rect, False_rec (後で、説明する)
  - case タクティックで使う (後で, 説明する)

論理結合子:論理積

自然演繹による導入: $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_i$

Coq で (NB: A と B は暗黙のパラメーター, 穴埋め _で推論できる):

型　パラメーター　ソート

```
Inductive and (A B : Prop) : Prop :=
  conj : A -> B -> A /\ B.
```

構成子 ⟶ conj

A /\ B: 構成子の型

and 型の定義によって, 次の導入が行われる:

- constants: and, conj
  - 例: conj I I : True /\True
    (NB: conj p q $\stackrel{def}{=}$ @conj P Q p q $\stackrel{def}{=}$ @conj _ _ p q) （スライド 116）
- and 型のデータ構造を消費するための elimination ルール:
  - and_ind : forall A B P : Prop, (A ->B ->P) ->A /\ B -> P
    - つまり, A と B で P を証明できれば, A /\ B でも証明できる
  - and_rect, and_rec
- (伝統的な) タクティック: split (apply conj より一般的) (NB: ただ, 伝統的な Coq なので, 仮定の自動導入を気を付けて)

論理結合子:論理和

自然演繹による導入:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_{i1} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_{i2}$$

Coq で:

型　パラメーター　ソート

```
Inductive or (A B : Prop) : Prop :=
| or_introl : A -> A \/ B
| or_intror : B -> A \/ B
```

構成子

構成子の型

or 型の定義によって, 次の導入が行われる:

- constants: or, or_introl, or_intror
  - 例: or_intror False I : False \/ True
- or 型のデータ構造を消費するための elimination ルール:
  - or_ind, or_rect, or_rec
- (伝統的な) タクティック: left, right (apply or_introl, apply or_intror の代わりに)

Outline

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, Q : Prop ===== P /\ Q -> Q /\ P	case.	P, Q : Prop ===== P -> Q -> Q /\ P

- case: H. $\stackrel{def}{=}$ move: H. case.
- 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

参考ファイル➡logic_example.v

case による証明

ゴール (前)	タクティック	ゴール (後)
P, Q : Prop ===== P /\ Q -> Q /\ P	case.	P, Q : Prop ===== P -> Q -> Q /\ P
証明 (前)		証明 (後)[4]
(fun P Q : Prop => ?3)		(fun (P Q : Prop) (H : P /\ Q) => match H with | conj H0 H1 => ?10 H0 H1 end)

参考ファイル ➡logic_example.v, exo4-7

[4]simplified; match 構文について後で説明する (とりあえず, and_ind だと理解すれば良い)

Outline

型付きプログラミング言語 Gallina の概要

- 項 (NB: 型を含む):

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

- 帰納的に定義された型の値は構成子 (つまり, constant) となり, `match` で消費する (再帰的で帰納的に定義された型なら, `fix` と組合せて使える)
- 注意: Coq 関数の停止性（スライド 69）

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 な同値関係は計算によって一致する項を同じもの（definitional equality ともいう）とみなす:

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

(NB: 型検査決定可能にするため, 停止性の保証が要ることが分る)

- 証明の一部は計算になる時に, リフレクションという (スライド 106)
- $=_{\beta\delta\zeta\iota}$ [CDT15, Sect. 4.3]:
  - β: β 簡約 (つまり, $(\texttt{fun}\, x \Rightarrow t_1)t_2 \to_\beta t_1\{t_2/x\}$)
  - 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 型を持つ単項述語を表す
  - Set の型を持つ型のデータ構造は discriminate できる ($0 \neq 1$)
  - 一般的に, Prop の型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) [CDT15, Sect. 4.5.4]
    - $\Rightarrow$ Proof irrelevance は admissible
- Large sorts(universe とも呼ぶ): $\mathtt{Type}_i$
  - ヒエラルキー:

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

  - 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:
  - ~~$\leq_{\beta\delta\zeta}$~~ $\to \leq_{\beta\delta\zeta\iota}$
  - Cumulativity: $\mathtt{Set} \leq_{\beta\delta\zeta\iota} \mathtt{Type}_i$; $\mathtt{Type}_i \leq_{\beta\delta\zeta\iota} \mathtt{Type}_j, i \leq j$
  - Prop $\leq_{\beta\delta\zeta\iota}$ Set, thus Prop $\leq_{\beta\delta\zeta\iota}$ $\mathtt{Type}_i$ [CDT15, Sect. 4.3]

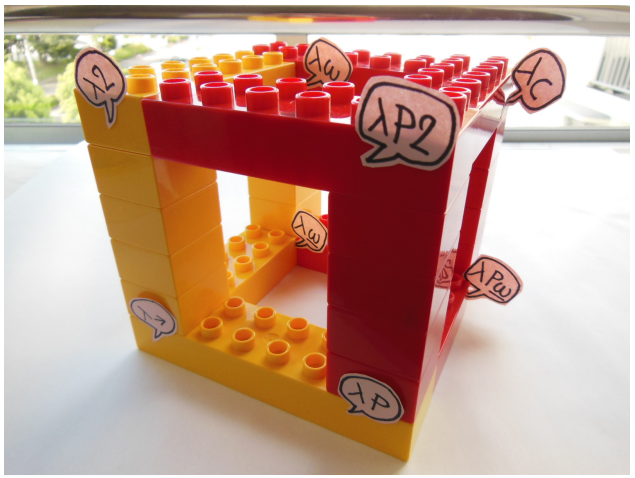
参考ファイル➡logic_example.v

Product の型付け規則

Var, Lam, App ルールに加えて,
product の構成のためのルール
(PTS (Pure Type System) の場合 [NG14]):

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \texttt{forall}\, x : A, B : s_2}\ \text{Prod}$$

(NB: λ cube の説明となる)



$\emptyset \vdash * : \Box$ を想定すると（Coq の場合: $* \approx$ `Prop`/`Set`, $\Box \approx$ `Type`）:

- $(s_1, s_2) = (*, *)$ or $(\Box, *)$ (polymorphism, impredicativity) $\rightarrow \lambda 2$ (= System F)
  - 型に依存する項
  - 例: $(\texttt{fun}\, \alpha : * \texttt{=>} \texttt{fun}\, x : \alpha \texttt{=>} x) : \texttt{forall}\, \alpha : *, \alpha \to \alpha$;　$\texttt{forall}\, \alpha : *, \alpha \to \alpha : *$
- $(s_1, s_2) = (*, *)$ or $(\Box, \Box)$ (型の構成) $\rightarrow \lambda\underline{\omega}$
  - 型に依存する型
  - 例（型の構成）: $(\texttt{fun}\, \alpha : * \texttt{=>} \alpha \to \alpha) : * \to *$;　$* \to * : \Box$
- $(s_1, s_2) = (*, *)$ or $(*, \Box)$ (依存型) $\rightarrow \lambda P$ (= $\lambda\Pi \simeq$ AUTOMATH)
  - s_1 は $\Box$ にならず、$A : *$ だけ、従って、x は項（項に依存する型）
  - 例（型の構成）: $\texttt{fun}\, n \texttt{=>} list_n : \mathbb{N} \to *$;　$\texttt{fun}\, n \texttt{=>} isprime_n : \mathbb{N} \to *$;　$\mathbb{N} \to * : \Box$
- $\lambda 2 + \lambda\underline{\omega} + \lambda P \rightarrow \lambda C$ (= $\lambda P\omega$ = CC $\simeq$ Coq)

依存型 (forall $x : A, B$ / $\prod_{x:A} B$) の構成 (1/3)

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

例:

- `True ->True : Prop` (Prop, Prop)
- `(forall x : nat, 0 <= x) : Prop` (Set, Prop)
- `(forall p : Prop, p -> p) : Prop` (Type, Prop) (polymorphism, impredicativity)
- `(forall x : nat, x = x) : Set` (quantification over datatypes)
- `(forall P : nat ->Prop, P 0 ->exists n : nat, P n) : Prop` (quantification over predicate types)

依存型 (forall $x : A, B$ / $\prod_{x:A} B$) の構成 (2/3)

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

例:

- `nat -> nat : Set` (Set, Set) (function types)
- `Fail Check (forall x : Set, x ->x) : Set.`
- No (Type, Set) ⇒ Predicative Calculus of Inductive Constructions (Coq v8 から)

依存型 (forall $x : A, B$ / $\prod_{x:A} B$) の構成 (3/3)

Predicative universes を含む Prod 型付け規則 (NB: Coq= CC_ω, CC with universes) :

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

例:

- `Prop -> Prop : Type`
- `Set -> Set : Type`
- `(forall x : Set, x -> x) : Type` (`Set` is predicative)
- `nat -> Prop : Type` (`Set`, `Type`) (first-order predicates)
- `(Prop -> Prop) -> Prop : Type` (higher-order polymorphism)
- `(forall P : nat ->Prop, Prop) : Type` (higher-order type)

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]:

$$\dfrac{\dfrac{\dots}{\vdash \texttt{Prop} : \texttt{Type}_1} \qquad \dfrac{\dfrac{\dots}{A : \texttt{Prop} \vdash A : \texttt{Prop}} \qquad \dfrac{\dots}{A : \texttt{Prop}, \_ : A \vdash A : \texttt{Prop}}}{A : \texttt{Prop} \vdash A \to A : \texttt{Prop}}\ \text{Prod-Prop}}{\vdash \texttt{forall}\, A : \texttt{Prop}, A \to A : \texttt{Prop}}\ \text{Prod-Type}$$

- Type は predicative:

$$\dfrac{\dfrac{\dfrac{\dots}{1 < 2}}{\vdash \texttt{Type}_1 : \texttt{Type}_2, 1 \le 2} \qquad \dfrac{\dfrac{\dots}{A : \texttt{Type}_1 \vdash A : \texttt{Type}_1} \qquad \dfrac{\dots}{A : \texttt{Type}_1 \vdash \texttt{Type}_1 \le_{\beta\delta\zeta\iota} \texttt{Type}_2}\ \text{Conv}}{A : \texttt{Type}_1 \vdash A : \texttt{Type}_2, 2 \le 2}\ \text{Prod-Type}}{\vdash \texttt{forall}\, A : \texttt{Type}_1, A : \texttt{Type}_2}$$

Prop impredicative / Type predicative

Coq で

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

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

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

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

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

DupTypeProof の型は Type にあるが,
Check (DupTypeProof _ DupTypeProof). は失敗する.
Coq 8.5 の universe polymorphism (Polymorphic)[ST14] で Check で成功

- 他の例:

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

参考ファイル➡predicative_example.v

依存型ペア (Dependent Pair)

- Reminder: 非依存型ペアは (a, b) と書く

```
Inductive prod (A B : Type) : Type :=
| pair : A -> B -> A * B.
Check (0:nat, True) : nat * Prop
```

- 依存型ペアは存在記号 $\exists a : A.P\,a$ で書く (紙上で $\Sigma_{x:A}.P\,x$ でも書く)
- Coq で依存型ペアはプリミティブではない; 帰納的型で記述する:

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

witness ↑ (x : A)
証明 ↑ (P x)
exists x, P x = ex P (witness も証明もない)

- 導入のルール: apply ex_intro (伝統的なタクティク: exists)
- 除去ルール: case タクティク

Dependent Pairs

- Dependent pair の定義の自由度:

	Aの型	Pの型	最終な型	記号
existential quant.	Type	Prop	Prop	exists x, P x
weak dep. sum	Type	Prop	Type	{x | P x}
strong dep. sum	Type	Type	Type	{x : A & P x}

- Weak dependent sum (a.k.a. subset type)
  - ```
    Inductive sig (A : Type) (P : A -> Prop) : Type :=
      exist : forall x : A, P x -> {x | P x}
    (* projections: proj1_sig, proj2_sig *)
    ```
  - 例えば, 3 より小さい自然数の集合 (Ordinal を参照, スライド 130) :

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

- 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 *)
```
```

Dependent Pair の応用

- Dependent record は dependent pair の一般化; 例えば:

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

数学の形式化に大事な役割を果す（telescopes [dB91], packed classes [GGMR09]）

- Dependent pair の応用例:
  - 写像 (“witness”: `s`, 言明: `ordered (unzip1 s)`):

```
Inductive map :=
| mkMap : forall s : seq (nat * bool), ordered (unzip1 s) -> map.
```

  - コンピュータの整数 (“witness”: `s`, 言明: `size s =n`):

```
Inductive int (n : nat) :=
| mkInt : forall s : seq bool, size s = n -> int n.
```

文字型: `int 8`, ポインタ: `int 32` 等

参考ファイル➡dependent_example.v, exo8-14

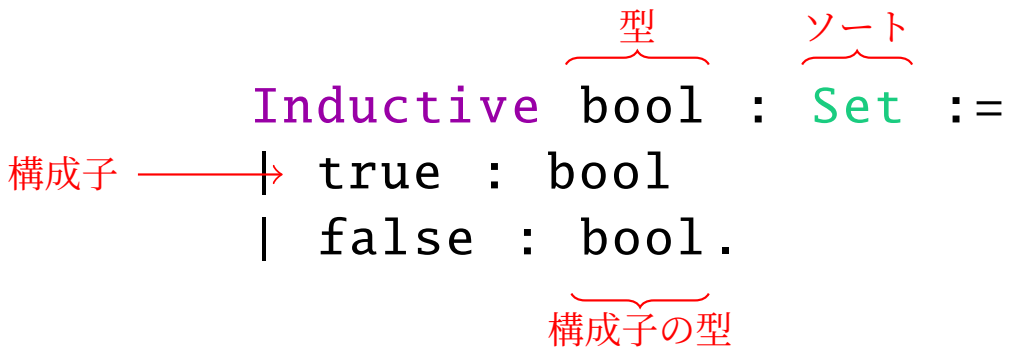
帰納的型で定義されている論理結合子の纏め

自然演繹 (導入ルールだけ)	Coq による帰納的型	タクティク	
		導入	除去
$\dfrac{}{\Gamma \vdash \mathrm{True}}\ \mathrm{T}_i$	`Inductive True : Prop :=` `| I : True.`	`apply I`	×
×	`Inductive False : Prop := .`	×	`case`
$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_i$	`Inductive and` `(A B : Prop) : Prop :=` `| conj : A -> B -> A /\ B.`	`split`	`case`
$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee_{i_L}$ $\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee_{i_R}$	`Inductive or` `(A B : Prop) : Prop :=` `| or_introl : A -> A \/ B` `| or_intror : B -> A \/ B`	`left` `right`	`case`
$\dfrac{\Gamma \vdash P[x := t]}{\Gamma \vdash \exists x, P\,x}\ \exists_i$	`Inductive ex (A : Type)` `(P : A -> Prop) : Prop :=` `| ex_intro :` `forall x : A, P x ->` `exists x, P x`	`exists`	`case`

Outline

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`
- ブール型のデータ構造を消費するための elimination ルール:
  - `bool_rect`: strong elimination (NB: `Type` の述語)
  - `bool_rec`: (NB: `Set` の述語) (`bool_rect` の特化)
  - `bool_ind`: (NB: `Prop` の述語) (`bool_rect` の特化; 論理的なリーゾニングのため, `case`/`elim` タクティックを参照)
- ι-簡約ルールの実現:

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

参考ファイル➡ssrbool_example.v

Elimination ルールによるプログラムの例

ブールを受け取って, 自然数を返す:

```
Definition nat_of_bool :=
  bool_rec (fun _ => nat) 1 0.
Check (nat_of_bool : bool -> nat).
```

ブールを受け取って, 自然数またはブールを返す (依存型を利用):

```
Definition dep_of_bool :=
  bool_rec (fun b => match b with true => nat | false => bool end) 1 true.
Check (dep_of_bool : forall b, match b with true => nat | false => bool end).
```

NB: bool_rect に頼らない記述:

```
Definition dep_of_bool2 b : match b with true => nat | false => bool end :=
  match b with
    | true => 1
    | false => true
  end.
```

参考ファイル➡ssrbool_example.v

帰納的（に定義される）型

再帰型

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

構成子の argument の型

- 自然数 nat の型は Peano 自然数を定義する
  - constants: nat, O, S
  - O (大文字の「オー」) は零, S O は 1, S (S O) は 2 等
- nat 型のデータ構造を消費するための elimination ルール:
  - nat_rect, nat_rec, nat_ind
  - 再帰的な関数の定義のプログラム:
    - Gallina の fix (無名の不動点) / Vernacular の Fixpoint (named 不動点)
  - 数学帰納法による証明
    - 帰納的な型の定義の際, Coq は帰納法の原理を自動生成する
    - elim タクティックを参照 (スライド 74)

参考ファイル➡ssrnat_example.v

数学的帰納法

- 関数の型として言明を記述する: 参考ファイル➡ssrnat_example.v

```
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) は帰納ステップのこと
  - forall n, P n は証明したいこと

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

fix/Fixpoint

- Coq での関数が停止しなければならない
  - そうでないと, 型検査の決定可能性がなくなり, `False` の証明も作れる：

```
        OCaml version 4.02.1

# let rec loop (n : int) = loop n;;
val loop : int -> 'a = <fun>
```

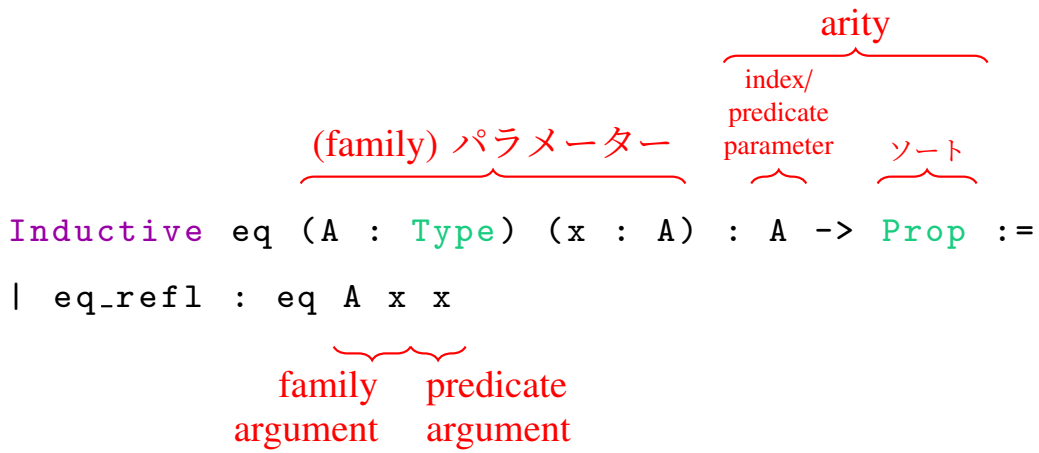
  - 従って, “strictly positive” な帰納的型しか許さない：

```
Fail Inductive term : Set :=
| abs : (term -> term) -> term.
```

    (NB: 定義中の帰納的型は, 構成子の argument の product の右側にしか表れない, HOAS 関係の研究に参考（例：[DFH95]）)
- Coq が構造的な再起呼出を自動的に認める（`struct` 構文; サブ項を見て、判断する）
- それ以外は証明が要る （やり方: [BC04, Chapter 15], 特に標準ライブラリの `Coq.Init.Wf` に参考）

Outline

帰納的型: 同値関係 (Propositional Equality)



- `Inductive eq` は型族 (family of inductive propositions) を定義する
  - 型 `A` と項 `x` はパラメーター, 三番目の引数は index
  - 同値関係は依存型である
  - `eq A x x` は型 (言明:同値関係は反射)
- 同値関係 $x =_A y$ は Coq で `x = y` と書ける (`eq A x y`, つまり A は推論される)
- 同値関係の構成子の型: `eq_refl : forall (A : Type) (x : A), x =x`
  Coq の `reflexivity` タクティクは `apply eq_refl` である

Propositional Equality → Leibniz Equality

- eq の除去ルール eq_rect:

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

- eq_ind は eq_rect のインスタンスである：

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

eq_ind があるから, eq は Leibniz equality とも言う

- 書き換えは除去ルール eq_ind によってできる

Outline

elim タクティック—帰納法

elim は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]. 参考ファイル➡tactics_example.v
- ユーザが帰納法の補題を選べられる (書き方: elim/myT_ind) スライド 76

Coq vs. SSReflect

Coq の induction の自動化は相応しくない場合は多いので, elim+move 等を優先的に使う

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)[5] `(nat_ind` `(fun n : nat =>` `n + n = 2 * n)` `?3 ?4)`

[5] simplified

帰納的型と帰納法：落とし穴

- Strong induction:

```
forall P : nat -> Prop, (forall m, (forall k, (k < m) -> P k) -> P m) ->
  forall n, P n.
```

  - Coq.Arith.Wf_nat の lt_wf_ind, SSReflect idiom
- Custom induction を手で帰納法を構成する
- 相互帰納的型：

```
Inductive tree (A : Set) : Set :=
  node : A -> forest A -> tree A
with forest (A : Set) : Set :=
  fnil : forest A
| fcons : tree A -> forest A -> forest A.
```

  Scheme 命令を用いて、帰納法を生成する
- 入れ子帰納的型：

```
Inductive ltree (A : Set) : Set :=
  lnode : A -> list (ltree A) -> ltree A.
```

  手で帰納法を構成する [BC04, Sect. 14.3.3]

rewrite タクティック

基本的な機能

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

- 同値関係の elimination ルールを利用
  - rewrite H は apply (eq_ind _ ... _) として考えていい
  - (NB: つまり, Coq では, apply を用いて, 含意の除去・理論結合子の導入・帰納法・書き換えを実現する)

参考ファイル➡tactics_example.v

rewrite 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）
n : nat n0 : n = 0 m : nat ===================== m + n = m	rewrite n0.	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)[6] (fun (n : nat) (n0 : n = 0) (m : nat) => eq_ind_r (fun n1 : nat => m + n1 = m) ?8 n0)

[6] simplified

rewrite タクティック

パターンの出現スイッチ

ゴール (前)	タクティック	ゴール (後)
`n : nat` `n0 : n = 0` `m : nat` `======================` `n + m = n`	`rewrite {1}n0.` または `rewrite {-2}n0.`	`n : nat` `n0 : n = 0` `m : nat` `======================` `0 + m = n`

- {1}は一番目のパターンを選ぶ
- {-2}は二番目以外全パターンを選ぶ
- (NB: 注意: 前のスライドと違うゴール)

rewrite タクティック

パターンの出現制限

ゴール (前)	タクティック	ゴール (後)
`x, y : nat` `H : forall t u : nat,` `t + u = u + t` `======================` `x + y = y + x`	`Fail rewrite {2}H.` `rewrite [y + _]H.`	`x, y : nat` `H : forall t u : nat,` `t + u = u + t` `======================` `x + y = x + y`

- `rewrite [H1]H.` はパターン H1 の中を書き換える
- `rewrite {1}H.` より確実
- (NB: 例は [GMT08] による)

rewrite タクティック

パターンのコンテキスト指定 [GT12]

ゴール (前)	タクティック	ゴール (後)
a, b, c : nat ========== a + b + 2 * (b + c) = 0	Fail rewrite {2}addnC. rewrite [b + _]addnC. rewrite [in 2 *_]addnC. rewrite [in X in 2 *X]addnC.	a, b, c : nat ========== a + b + 2 * (c + b) = 0

- rewrite [in X in ...X...]H.: ゴールを...X... とマッチして, その X の中を書き換える

rewrite タクティック

その他の機能

- 逆スイッチ:

`rewrite H.`	左から右へ	`rewrite -H.`	逆の書き換え
`rewrite /mydef.`	定義の展開	`rewrite -/mydef.`	folding

- multiplicity スイッチ:

`rewrite n!H.`	n回書き換え	`rewrite ?H.`	0 回以上書き換え
`rewrite !H.`	一回以上書き換え	`rewrite n?H.`	n 回以下書き換え

- 連続の書き換え
  - `rewrite H1 H2.` $\stackrel{def}{=}$ `rewrite H1; rewrite H2.`
  - `rewrite (_ : lhs =rhs).` は Coq の `cutrewrite` と同じ効果
- ⟨*s-item*⟩ (“simplification operation”)
  - `rewrite //.` は `try done.` と同じ効果
  - `rewrite /=.` は `simpl.` と同じ効果 (NB: `//=` $\stackrel{def}{=}$ `// /=`)
- clear スイッチ:
  - `rewrite {H}.` は `clear H.` と同じ効果
- `move` との関係
  - ⟨*s-item*⟩, clear スイッチ, 逆スイッチ, 出現スイッチは `move` でも使える
  - `move=>->.` $\stackrel{def}{=}$ `intro TMP; rewrite TMP; clear TMP.`

Outline

C コンパイラ (CompCert) I

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; イメージ:

```
Definition compcert := fun (c : C_prg) => (t : ASM_prg).
Lemma correctness : forall c o, observe o c -> observe o (compcert c).
```

- 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)
- 2004 年から INRIA で X. Leroy らが形式検証を続けている [Ler09, BL09]
- 2013 年: X. Leroy は Microsoft Research Verified Software Milestone 賞を受賞
- 約 50,000 行のスクリプト, 4 人年だと言われている
- 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 (CompCert) II

- 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」

- 最新の成果: Verasco's abstract interpreter (形式検証済みの自動解析); 受賞：Royal Society Milner Award 2016

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), JavaCard API 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

- NB: 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]

seL4 マイクロカーネル

- オーストラリアの NICTA 研究所
- 定理証明支援系: Isabelle/HOL
- seL4 のソースコード: C 言語; 約 8,700 行
- アプローチ:
  - ホーア論理
  - (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

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 の構文解析機能と Ltac によって実現 [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]
- 新しいライブラリ
  - 四色定理 → SSReflect; 奇数位数定理 → MathComp
  - Coq の標準タクティックと Coq の標準ライブラリはまだ使える
- タクティックの向上
  - タクティックの数を減少, タクティカル等によって一般化 (特に, `move` と `rewrite`)
    - 小規模リフレクション (スライド 106)
  - メンテナンスのため: スクリプトの構造化 (スライド 95), 名付けることの強制

⇒ スクリプトと証明は短くなる, スクリプトは robust になる

Coq と SSReflect の関係

Gallina と Vernacular は殆ど変更なし

“新シンタクス”:

- 帰納的な型. 例: Coq のリスト：

```
Inductive list (A : Type) : Type := nil | cons : A -> list A -> list A.
```

SSReflect のシンタクス:

```
Inductive seq (A : Type) : Type := nil | cons of A & seq A.
```

- pattern testing. 例: Coq で, s の全要素は述語 a を満す：

```
Variables (T : Type) (a : T -> bool).
Fixpoint myall s := match s with
                      x :: s' => a x && myall s'
                    | _ => true
                    end.
```

SSReflect のシンタクス:

```
Fixpoint all s := if s is x :: s' then a x && all s' else true.
```

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

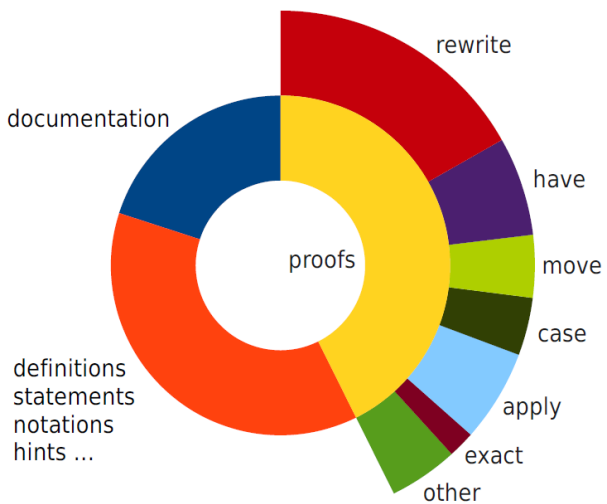
上記の状況は欠点と見做すべきではないが...

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

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 [7]: `Move`, `Case`, `Apply`, `Step`[8], `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)



[7] Coq v.7 のころ, タクティック名の先頭は大文字だった
[8] 今の `have` (Coq の `assert` に似ている)

Proof Engineering

数個のタクティックで数万行のスクリプトを書く ...

証明スクリプトの作成による問題はプログラミングと同じ:

- 変数等の名前を真面目に選ぶことは重要
  - “Search for theorem names is a major difficulty.” [Hal12]
  - 一定の命名規則を守る (SSReflect の強みの一つ)
  - 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
    - 変数と仮定の名前は本当にどうでもいいなら次を使う:
      `move=>?.` トップをポップするが, その変数または仮定を指すことができない
      `move=>*.` `move=>?`を繰り返す
- コピペしない
  - その代わりに, `set` タクティックを使う (`rewrite` のためのような contextual pattern あり [GMT08, Sect. 8.3.1])
- 記号を効率的に使う (記号は抽象化のための道具として考える)
- 証明が大きくなると, メンテナンスは重要になる
  - 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（どうせ絶対いつか壊れるので、失敗するところを攻めやすくする）

Structured スクリプト

証明が大きくなると, スクリプトの壊れるところを攻めるように

- スクリプト? backward reasoning を挟んだ forward reasoning として考えていい (スライド 101)
- スクリプトの木構造を明かにする:
  - +, -, *: 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 $\stackrel{def}{=}$ 成功しなければ先に進まないタクティック
      (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
- 証明ステップ: ~~一つのタクティック~~ → 一行のスクリプト

Coq vs. SSReflect

SSReflect の `by`, Coq で, `now` という（既に `by` があったから）

Structured スクリプト

具体的な例

```
Lemma undup_filter {A : eqType} (P : pred A) (s : seq A) :
  undup (filter P s) = filter P (undup s).
Proof.
elim: s => // h t IH /=.
case: ifP => /= [Ph | Ph].
- case: ifP => [Hh | Hh].
  + have : h \in t.
      move: Hh; by rewrite mem_filter => /andP [].
    by move=> ->.
  + have : h \in t = false.
      apply: contraFF Hh; by rewrite mem_filter Ph.
    move=> -> /=; by rewrite Ph IH.
- case: ifP => // ht.
  by rewrite IH /= Ph.
Qed.
```

参考ファイル➡tactics_example.v, exo17

(NB: ifP ⇒ スライド 123, have ⇒ スライド 101, /andP ⇒ スライド 107, \in ⇒ スライド 129)

Outline

congr タクティック

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' ```

参考ファイル➡tactics_example.v

Coq vs. SSReflect

congr は f_equal を一般化 (NB: かつ、Coq8.5beta まで f_equal の実装は欠陥がある)

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
- (NB: `set` と違う (定義/展開 vs. 同値関係/書き換え))

等式の生成 + case

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

ゴール (前)	タクティック	ゴール (後)
a, b : nat ======== a <> b	case H : a =>[|n].	... H : a = 0 ============ 0 <> b ... n : nat H : a = n.+1 ============ n.+1 <> b

- スライド 125 も参照

Coq vs. SSReflect

等式の生成は Coq の case_eq でもできる

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, `move=>[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

Coq vs. SSReflect

`have` は `assert` を一般化

wlog タクティック

- wlog = “without loss of generality”
- 具体的な “例”:

```
Variable a : nat.
Hypothesis a1 : 1 < a.

Lemma artificial (k l : nat) : k < l \/ l < k -> a ^ k != a ^ l.
Proof.
wlog : k l / k < l.
  move=> H.
  case=> kl.
    apply H => //; by left.
  rewrite eq_sym.
  apply H => //; by left.
move=> kl _.
by rewrite eqn_exp2l // neq_ltn kl.
Qed.
```

参考ファイル➡tactics_example.v

Outline

move + ビュー (view)

move/H. はトップ仮定を変形する (≃ “apply on-the-fly”)

ゴール (前)	タクティック	ゴール (後)
P, Q : Prop PQ : P -> Q ====================== P -> Q	move/PQ.	P, 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, ビュー補題が適用される; 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)

Coq vs. SSReflect

move/(_ a b c) はトップを特化し, Coq の specialize に当る

参考ファイル ➡ view_example.v, exo18

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, Q : Prop PQ : P <-> Q p : P ===== Q	apply/PQ.	P, 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` を満たす関数 `f` 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 52)
  - 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う
  - `Prop` で deduction する (例えば, `P \/ Q` に対して `case` で場合分けをする; `a = b` に対して, `rewrite` で書き換える)
  - 決定的な時に単計算で証明負担を減らす (例えば, `b1 || b2` に対して `case: b1` と `case: b2` で, 真理値表を使う; 同値関係は計算で決める)
  - `bool` を重視するために, `is_true` コアーションを使う. 具体的に, SSReflect で `forall P : bool, P -> P` は次の略になっている: `forall P : bool, P = true -> P = true`

ビューとブールリフレクション

andb/&&(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)
```

(reflect 述語は<->として考えていい)

ゴール（前）	タクティック	ゴール（後）
P, Q : bool ===== P && Q -> Q (* P && Q = true -> Q = true *)	move/andP.	P, 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, Q : bool ===== P && Q -> Q	case/andP.	P, 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, Q : bool` `======================` `P || Q -> P || Q`	`case/orP.`	`P, 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].`

参考ファイル ➡ view_example.v, exo19-21

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, m : nat` `==========` `eqn n m -> n = m`	`move/eqnP.`	`n, 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 124)

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 wlog
simpl	move=>/= rewrite /=
clear H	{H}
elim induction	elim
now	by
discriminate	done
assumption	move=>//
contradiction	rewrite //
pattern	rewrite [...]H set
f_equal	congr

Outline

SSReflect/MathComp 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, 暗黙の引数の使い方, 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
`About`	`Check`より情報量は多い
`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 Implicit`	暗黙の引数を見せる/見せない
`Set/Unset Printing All`	全部見せる/デフォルトに戻る

  - 通常の使い方に従わないと大変になる (特に, 定義の展開は一般的に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

SSReflect/MathComp ライブラリの補題を検索

補題の結論でフィルター? ⇒ 一番目のパラメーターにパターン; 例:

```
Search (_ < _)%N.
Search (_ < _ = _)%N. (* rewriting rule *)
```

パターンや記号や名前や文字列で結論と仮定をフィルター? ⇒ 二番目以上のパラメーター; 例:

```
Search _ (_ <= _)%N.
Search _ (_ <= _)%N "-"%N.
Search _ (_ <= _)%N "-"%N addn.
Search _ (_ <= _)%N "-"%N addn "add".
```

検索の範囲に制限? ⇒ モジュールを指定; 例:

```
Search _ (_ <= _)%N "-"%N addn "add" in ssrnat.
```

参考ファイル➡tactics_example.v

SSReflect/MathComp ライブラリの使い方

- `Require Import` の順番は使うファイルから習う
  - Coq8.5beta から, ライブラリの名前の混乱を避けるため、
    `From Ssreflect Require Import ssreflect` または
    `Require Import Ssreflect.ssreflect` を使うようになった
- 暗黙の引数 [CDT15, Sect. 2.7] の使い方に従うために, 次の Vernacular で初まる:

```
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

暗黙の引数の管理は重要なので, 次のスライドで上記の魔法を説明してみる

暗黙の引数の宣言の概要

参考ファイル➡implicit_example.v

- 関数を使う際コンテキストまたはその関数の一部の引数を用いて推論できる引数は「暗黙の引数」と言う. 例:ポリモーヒック関数
- 暗黙の引数の入力を省略するように Gallina の関数の宣言できる.
  { ... }で宣言するパラメータは暗黙の引数となる. 例:

```
Definition id_implicit {A : Type} (a : A) : A := a.
Check (id_implicit 0).
```

(NB: 明示的に宣言したパラメータを暗黙にするように Arguments の Vernacular 命令を使う.)

- 暗黙の引数を明示的にするよう「@」を使う. 例:

```
Check (@id_implicit nat 0).
Check (@id_implicit _ 0).
```

厳密な暗黙の引数

- 関数の一部の引数の型から必ず推論できる暗黙の引数は「厳密な暗黙の引数」と言う. 例, 下記の A は厳密な暗黙の引数:

```
cons : forall A : Type, A -> list A -> list A
```

- 下記の A は厳密な暗黙の引数であるが, l は厳密な暗黙の引数でない:

```
Set Implicit Arguments.
Inductive Pair (A : Type) := mkPair :
  forall l : list A, length l = 2 -> Pair A.
```

length l =2 の証明を与えても、必ず l を推論できるわけではない:

```
About mkPair.
Check (@mkPair _ (cons 0 (cons 0 nil)) (eq_refl (length (cons 0 (cons 0 nil)))
Check (@mkPair _ (cons 0 (cons 0 nil)) (eq_refl 2)).
Check (mkPair _ (eq_refl (length (cons 0 (cons 0 nil))))).
Fail Check (mkPair _ (eq_refl 2)).
```

厳密でない暗黙引数の推論と表示

厳密でない暗黙引数の推論を強制する：

```
Set Implicit Arguments. Unset Strict Implicit.
Inductive Pair (A : Type) := mkPair :
  forall l : list A, length l = 2 -> Pair A.
About mkPair.
Definition Pair00 := mkPair (eq_refl (length (cons 0 (cons 0 nil)))).
Print Pair00.
```

(NB: `A` と `l` は暗黙引数になるが, `Pair00` の表示の際表れる)

厳密でない暗黙引数の表示をしない：

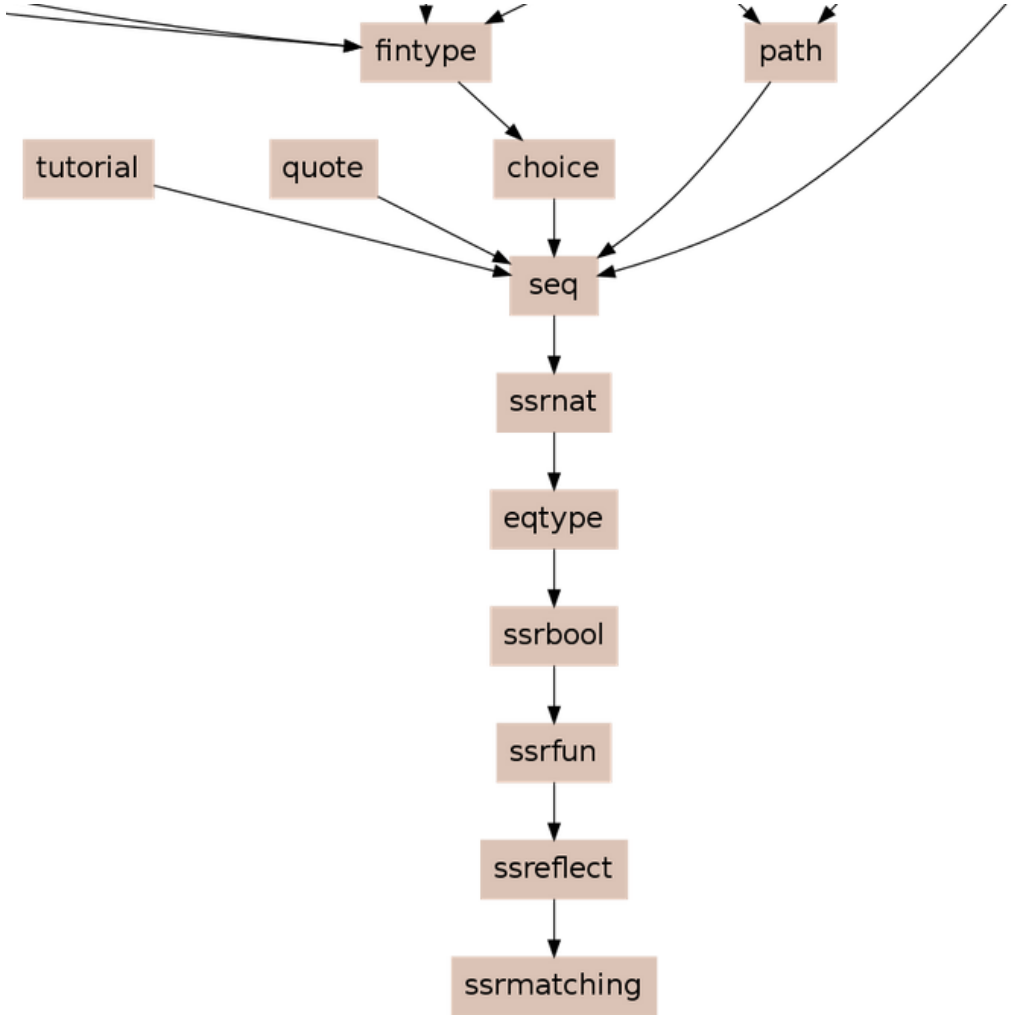
```
Unset Printing Implicit Defensive.
Print Pair00.
```

纏め:ssreflect/MathComp のファイルは次のヘッダーの説明

厳密な暗黙の引数を自動的に宣言するように	`Set Implicit Arguments.`
全ての暗黙の引数を自動的に宣言するように	`Unset Strict Implicit.`
厳密でない暗黙の引数を表示しないように	`Unset Printing Implicit Defensive.`

Outline

基礎ライブラリ



図は http://ssr.msr-inria.inria.fr/~jenkins/current/index.html 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 / aF x >-> rR x}`	`f (aF x) = rF (f x)`
`{morph f : x y / aOp x y >-> rOp x y}`	`f (aOp x y) = rOp (f x) (f y)`

ssrfun.v definitions

`injective f`	`forall x1 x2, f x1 = f x2 -> x1 = x2`
`cancel f g`	`g (f x) = x`
`involutive f`	`cancel f f`
`left_injective op`	`injective (op^~ x)`
`right_injective op`	`injective (op y)`
`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)`
`left_loop inv op`	`cancel (op x) (op (inv x))`
`self_inverse e op`	`op x x = e`
`commutative op`	`op x y = op y x`
`idempotent op`	`op x x = x`
`associative op`	`op x (op y z) = op (op x y) z`

- 全ファイルで使われている
- Search の検索に役に立つ

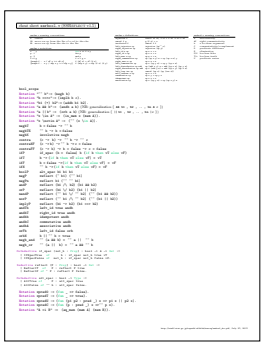
ssrbool.v の重要性

- ブールリフレクションの実現
  - ブール論理の記号 (例: &&, ||, ~~, ==>) と補題
  - コアーションによって, ブール値から Prop への埋め込み:

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

    ⇒ bool は Prop に見える
  - reflect 述語: ブールの世界と Prop の世界の等価性 (例: andP, orP, negP, implyP)
- pred 述語 (pred T $\stackrel{def}{=}$ T ->bool)
  - t が P を満すは P t と書く (t \in P と書ける時に “collective” 述語という)
  - リスト (スライド 129), 有限集合 (スライド 140) 等で使う
- 規則的な命名 (他のライブラリファイルは似た命名規則を使う)

参考資料: ssrbool_doc.pdf



例えば:

```
andTb   left_id true andb
andbT   right_id true andb
andbb   idempotent andb
andbC   commutative andb
andbA   associative andb
```

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 126):

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

次の帰納的型で実現:

```
CoInductive if_spec (A : Type) (b : bool) (vT vF : A) (not_b : Prop)
  : bool -> A -> Set := ...
```

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 110)
- 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?
  - move/eqP で, Leibniz 同値関係とブール値等式を変換
  - ==の仮定 H の書き換え: rewrite (eqP H)
- 決定可能な同値関係がある場合, (Leibniz) 同値関係の証明は一つしかないと Coq で証明できる (unicity of identity proofs); 例えば, ブールの場合:

```
forall (bool : Type) (x y : bool) (p1 p2 : x = y), p1 = p2
```

(スライド 130)

参考ファイル➡eqtype_example.v, exo22

例: ssrbool.v の boolP

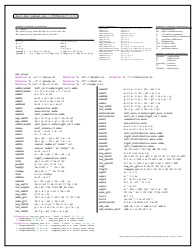
- `case H : p` は仮定 `H : p = true` と `H : p = false` を生成する (スライド 99)
- ライブラリを効率的に利用できるよう, 記号 (!=等) を使いたい
- 例えば (ここで, `ssrnat.v` を使う (スライド 126)):

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`, `exo23`

ssrnat.v: 自然数のための SSReflect 記号と定義

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 110)
  - 一定の命名規則
- 定義済のデータ構造と関数の再利用を重視; 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, 変形されないように, 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 (型族 (family of inductive propositions, n はパラメーター, 二番目の引数は index):

```
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 が多い

参考ファイル➡ssrnat_example.v, exo24-27

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, exo28-30

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

- A : eqType, s : seq A, a : A の場合, a \in s が書ける
  - 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:

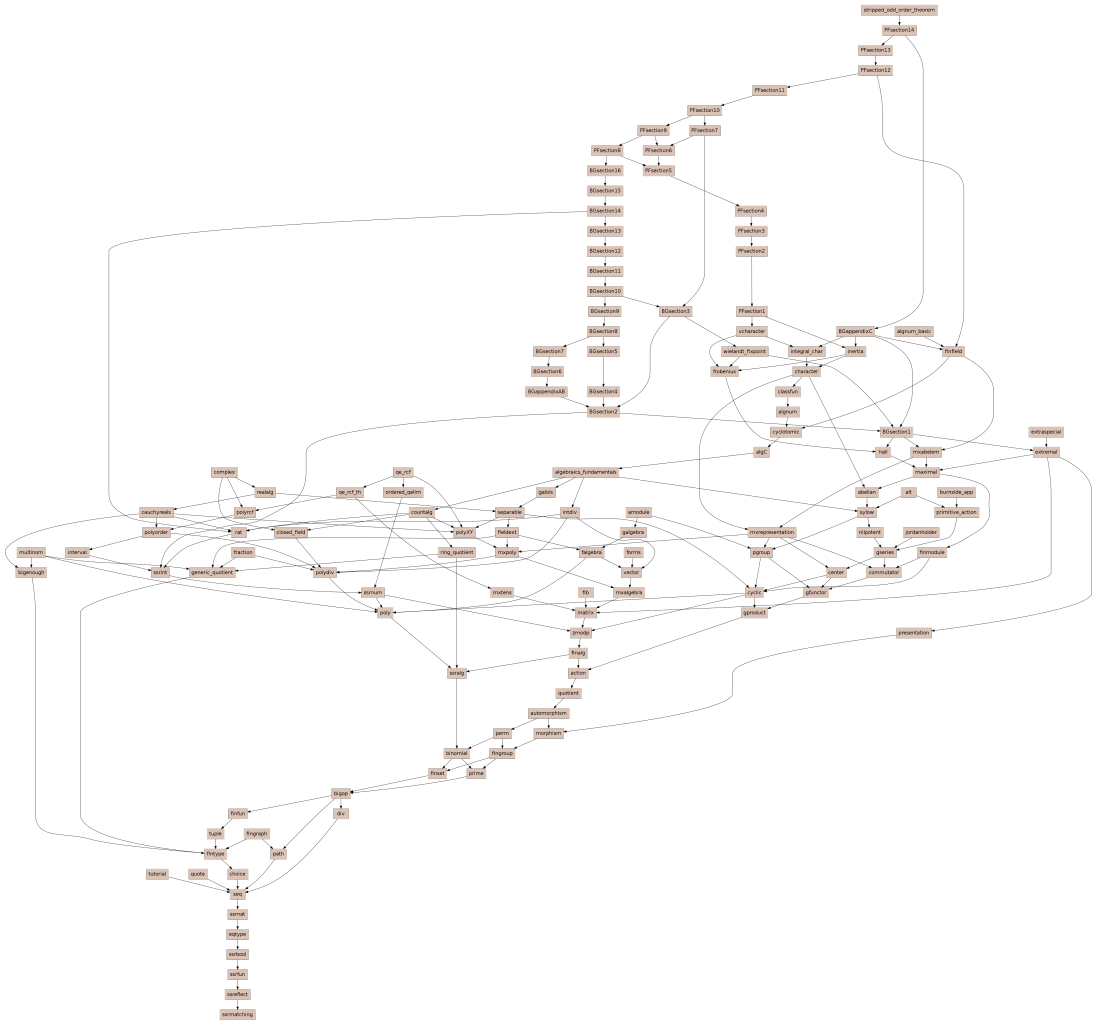
```
Variables (T : eqType) (a : pred T).
Fixpoint all s := if s is x :: s' then a x && all s' else true.
Lemma allP s : reflect (forall x, x \in s -> a x) (all a s).
```

fintype.v: 有限な数の要素のある型

- 要素のリストを取り出す:
  - T : finType なら, T の要素のリストは enum T と書く
  - T : finType, P : pred T なら, 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 (スライド 61)
  - 上記の P はブール値なので, 同値関係の証明があり, $P\,x = true$ の証明は 1 つしかない (スライド 124)
  - 従って, 2 つの'I_n 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)

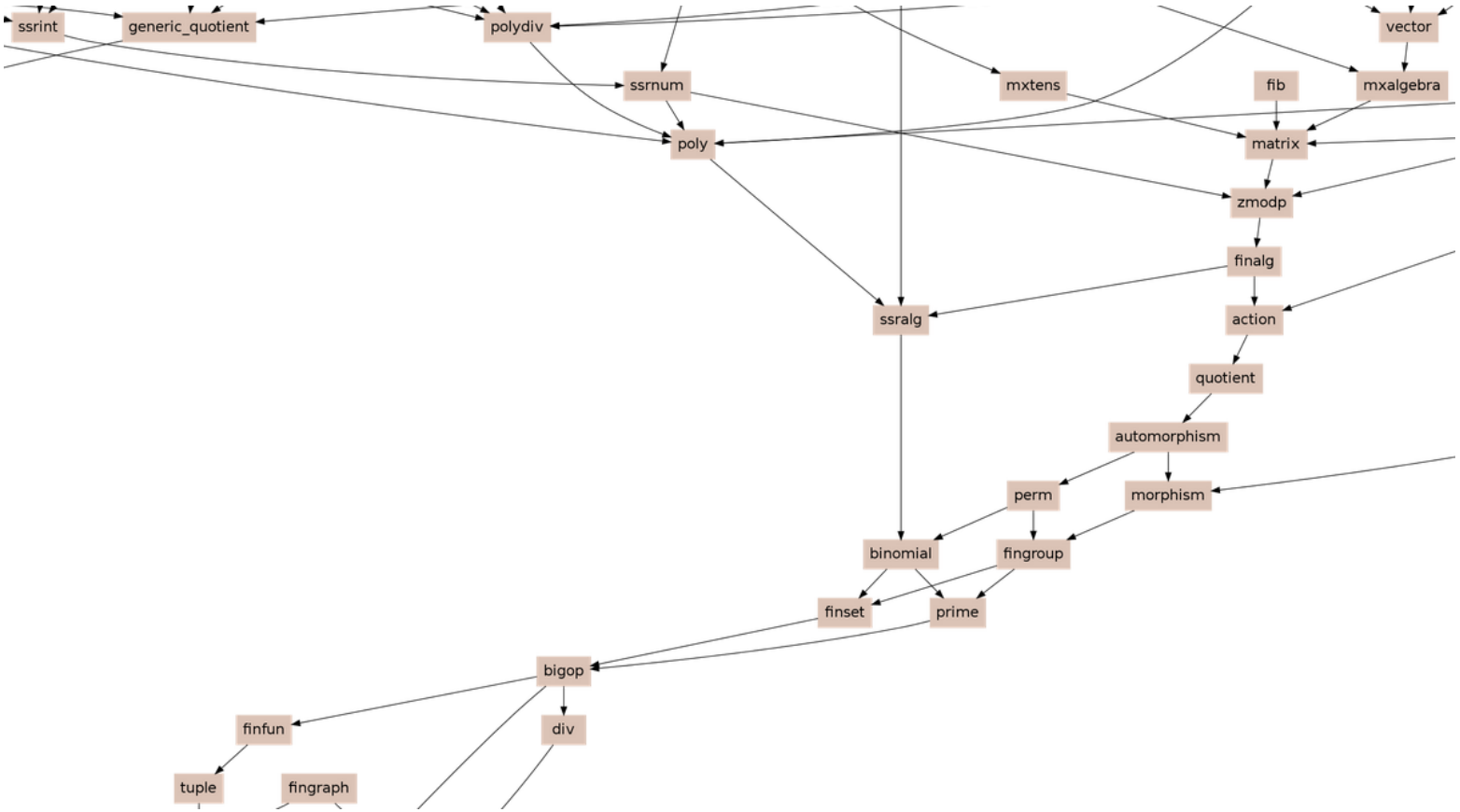
参考ファイル➡fintype_example.v

MathComp ライブラリ



図は http://ssr.msr-inria.inria.fr/~jenkins/current/index.html より

MathComp ライブラリ (拡大)



図は http://ssr.msr-inria.inria.fr/~jenkins/current/index.html 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が必要

- MathComp 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}.
```

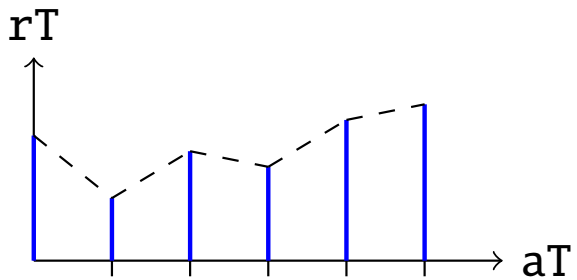
- 記号:
  - 型: n.-tuple T
  - 値: [tuple of s]; 例えば:
    - [tuple of [:: 1; 2; 3]]
    - [tuple of [seq x * 2 | x <- [:: 1; 2; 3]]]

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 (bigop_example.v で例がある):

```
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
```

Outline

bigop.v

- 数学と計算機科学 [GKP94, Chapter 2] に不可欠なライブラリ
  - 奇数位数定理の成功に大事なライブラリ [BGBP08]
- iterated operations: $+ \to \sum$, $\times \to \prod$ 等 ($\cup \to \bigcup$, $\cap \to \bigcap$: スライド 140)
- 記号 (自然数の場合):

紙上	MathComp (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`

- index はリストである
  - `index_iota` (区間)
  - `index_enum` (`'I_n`, `{ffun aT ->rT}`等の `fintype`)

bigop.v の基本

- 一般的な定義を提案する：

```
Variables (R I : Type) (op : R -> R -> R) (idx : R)
  (r : seq I) (P : pred I) (F : I -> R).
Definition bigop :=
  foldr (fun i x => if P i then op (F i) x else x) idx r.
```

- 一般的な補題を証明する：

```
Lemma bigop_split r1 r2 : bigop R I op idx (r1 ++ r2) P F =
  op (bigop R I op idx r1 P F) (bigop R I op idx r2 P F).
```

- インスタンスの詳細を記号で隠す：

```
Notation "\sum_ ( 0 <= i < n ) F" :=
  (bigop _ _ addn 0 (iota 0 n) xpredT (fun i => F))
  (at level 41, i at next level, format "\sum_ ( 0  <=  i  <  n )  F").
```

- どんなインスタンスでも同じ一般的な補題を使える；例えば：

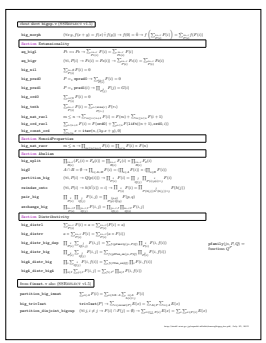
```
Lemma gauss n : 2 * (\sum_(0 <= i < n.+1) i) = n * n.+1.
```

参考ファイル➡mybigop_example.v, exo31-33

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 exo34 : 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 exo35 n : (6 * \sum_(k < n.+1) k ^ 2)%nat = n * n.+1 * (n.*2).+1.

  Lemma exo36 (x n : nat) : 1 < x -> (x - 1) * (\sum_(k < n.+1) x ^ k) =
  x ^ n.+1 - 1.

  Lemma exo37 (v : nat -> nat) (v0 : v 0 = 1) (vn : forall n, v n.+1 = \sum_(k <
  n.+1) v k) (n : nat) :
    n != 0 -> v n = 2 ^ n.-1.

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

  ただし, bigA_distr_big ($\prod_{i\in I}\sum_{j\in J}F(i,j)=\sum_{f\in J^I}\prod_{i\in I}F(i,f(i))$) を使う

参考ファイル➡bigop_example.v, exo31-34

finset.v: 有限集合

- T 型 (finType) 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}.
```

例えば:

```
Goal FinSet [ffun x : 'I_3 => true] = setT. ... Qed.
```

- 型: {set T}; 値: [set x | P]
- 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)].
```

- s : {set T}の場合, t \in s を書ける
  - set_type を predType として登録
- 外延性: Lemma setP A B : A =i B <-> A = B.

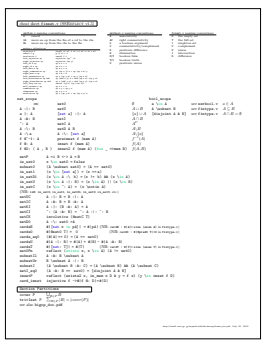
参考ファイル➡finset_example.v, exo38-40

finset.v: 有限集合

- bigop.v との関係:

紙上	MathComp (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`

参考資料: finset_doc.pdf



- 記号 (NB: 一部, ssrbool.v, fintype.v, bigop.v), 補題:

bigop.v: 応用例

- 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

  ⇒ dependent record を使う:

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

参考ファイル ➡bigop2_example.v

bigop.v: 確率分布の構築

- `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)
  - $\sum_{a \in A} \sum_{b \in B} P_1(a)\,P_2(b) = 1$? (`pair_big` 補題)
  - $\sum_{a \in A} \sum_{b \in B} P_1(a)\,P_2(b) = \sum_{a \in A} P_1(a)$? (確率分布の定義)
  - $\sum_{b \in B} P_1(a)\,P_2(b) = P_1(a)$? (`eq_bigr` 補題)
  - $P_1(a) \sum_{b \in B} P_2(b) = P_1(a)$? (`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)

参考ファイル ➡ `bigop2_example.v`

Outline

fingroup.v: 群の定義

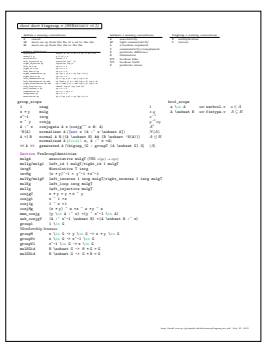
群は有限集合を用いて作る [GMR+07, BMR+, Tas14]

- `finGroupType` は group laws を持つ型である
- `gT : finGroupType` なら, `{group gT}`は群の型となる
- 群の定義は有限集合に基く
  - 基本的に, 群は次の命題を満たす集合 `A : {set gT}`である:

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

参考ファイル➡group_example.v

参考資料: fingroup_doc.pdf⬇



fingroup.v: 正規化群

- 共軛作用 (記号: x1 ^ x2):

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

- (集合の) 共軛作用 (記号: A :^ x):

```
Definition conjugate A x := 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)).
```

例 (参考ファイル➡group_example.v):

```
Variables (H G : {group gT}). Hypothesis HG : H <| G.
Lemma normal_commutes : H * G = G * H. Proof. ...
```

perm.v: 置換群

- 紙上での記号:
  - 置換の例: $(021) \stackrel{def}{=} (0 \mapsto 2; 2 \mapsto 1(; 1 \mapsto 0))$
  - 置換群 S_3 の例:

$\cdot \times \cdot$	(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)

- MathComp での例 (参考ファイル ➡permutation_example.v):
  - 置換群は'S_3 と書く
    - `Lemma S_3_not_commutative : ~ commute p01 p021.`
    - `Lemma card_S_3 : #|[group of <<[set x in 'S_3]>>]|=6.`
  - A_3 は $\{(012), (021), 1\}$ から生成された'S_3 の部分群:

```
Definition A_3 : {group 'S_3} := [group of << [set p021; p012; 1] >>].
Lemma group_set_A3 : group_set [set p021; p012; 1].
Lemma card_A_3 : #| A_3 | = 3.
```

  - A_3 は正規である

```
Lemma A_3_S_3 : A_3 <| [group of << [set x in 'S_3] >>].
```

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
- 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.
```

- 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

matrix.v: 例

符号理論による例 (参考ファイル ➡matrix_example.v):

- チェックシンボル行列:

```
Variables (l d : nat).
Variable A : 'M['F_2]_(l - d, d).
```

- パリティ検査行列:

```
Hypothesis dl : d <= l.
Definition H : 'M_(l - d, l) :=
  castmx (erefl, subnKC dl) (row_mx A 1%:M).
```

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

- 符号化:

```
Definition G : 'M_(d, l) :=
  castmx (erefl, subnKC dl) (row_mx 1%:M (-A)^T).
```

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

matrix.v: 例

ssralg.v と線型代数を使った例:

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

- $\left(\begin{array}{c|\|c} 1 & (-A)^T \end{array}\right) \times \left(\begin{array}{c|\|c} A & 1 \end{array}\right)^T = 0?$ (ゴール)
- $\left(1 \| (-A)^T\right) \times \left(\begin{array}{c} A^T \\ \hline\hline 1^T \end{array}\right)?$ (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)
- $(A - A)^T = 0?$ (linearD 補題)
- $0^T = 0?$ (addrN 補題)
- $0 = 0?$ (trmx0 補題)

Outline

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.

[Aff14a] ———, 定理証明支援系 *Coq* による形式検証, 名古屋大学 大学院多元数理科学研究科 理学部数理学科, 集中講義, Dec. 2014, `https://staff.aist.go.jp/reynald.affeldt/ssrcoq`.

[Aff14b] ———, 定理証明支援系 *Coq* 入門, 日本ソフトウェア科学会 第 31 回大会, チュートリアル, Sep. 2014, `http://jssst2014.wordpress.com/events/coq-tutorial/`.

[Aff14c] ———, 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.

[AG15] Reynald Affeldt and Jacques Garrigue, *Formalization of error-correcting codes: from hamming to modern coding theory*, Proceedings of the 6th International Conference on Interactive Theorem Proving, ITP 2015, Nanjing, China, August 24–27, 2015, Lecture Notes in Computer Science, vol. 9236, Springer, 2015.

[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.

[AK02] Reynald Affeldt and Naoki Kobayashi, *Formalization and verification of a mail server in Coq*, Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium, ISSS 2002, Tokyo, Japan, November 8–10, 2002, Revised Papers, 2002, pp. 217–233.

[AK08] ———, *A Coq library for verification of concurrent programs*, Electr. Notes Theor. Comput. Sci. **199** (2008), 17–32.

[AKY05] Reynald Affeldt, Naoki Kobayashi, and Akinori Yonezawa, *Verification of concurrent programs using the Coq proof assistant: a case study*, IPSJ Transactions on Programming **46** (2005), no. 1, 110–120.

[Alt93] Thorsten Altenkirch, *Constructions, inductive types and strong normalization*, Ph.D. thesis, University of Edinburgh, 1993.

[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.

[Bar10] Bruno Barras, *Sets in coq, coq in sets*, J. Formalized Reasoning **3** (2010), no. 1, 29–48.

[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.

[CDT15] ———, *The Coq proof assistant reference manual*, INRIA, 2015, Version 8.5beta2.

[CH84] Thierry Coquand and Gérard Huet, *A theory of constructions (preliminary version)*, International Symposium on Semantics of Data Types, Sophia-Antipolis, 1984, Jun. 1984.

[CH85] _____, *Constructions: A higher order proof system for mechanizing mathematics*, Proceedings of the European Conference on Computer Algebra, Linz, Austria EUROCAL 85, April 1–3, 1985, Linz, Austria, vol. 1 (Invited Lectures), Apr. 1985, pp. 151–184.

[CH86] _____, *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.

[CP90] Thierry Coquand and Christine Paulin, *Inductively defined types*, Proceedings of the International Conference on Computer Logic (COLOG-88), Tallinn, USSR, December 1988, Lecture Notes in Computer Science, vol. 417, Springer, 1990, pp. 50–66.

[dB91] N. G. de Bruijn, *Telescopic mappings in typed lambda calculus*, Inf. Comput. **91** (1991), no. 2, 189–204.

[dB03] _____, *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.

[DFH95] Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz, *Higher-order abstract syntax in coq*, Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications Typed Lambda Calculi and Applications, TLCA 1995, Edinburgh, UK, April 10–12, 1995, Lecture Notes in Computer Science, vol. 902, Springer, 1995, pp. 124–138.

[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.

[Geu95] Herman Geuvers, *A short and flexible proof of strong normalization for the calculus of constructions*, International Workshop on Types for Proofs and Programs, TYPES 1994, Båstad, Sweden, June 6–10, 1994, Selected Papers, Lecture Notes in Computer Science, vol. 996, Springer, 1995, pp. 14–38.

[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).

[GN91] Herman Geuvers and Mark-Jan Nederhof, *Modular proof of strong normalization for the calculus of constructions*, J. Funct. Program. **1** (1991), no. 2, 155–189.

[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.

[Hal13] ———, *Mathematics in the age of the turing machine*, arXiv:1302.2898v1 [math.HO], Feb. 2013.

[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.

[Luo90] Zhaohui Luo, *An extended calculus of constructions*, Ph.D. thesis, University of Edinburgh, 1990.

[MA08] Nicolas Marti and Reynald Affeldt, *A certified verifier for a fragment of separation logic*, Computer Software **25** (2008), no. 3, 135–147, Iwanami Shoten.

[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.

[MW03] Alexandre Miquel and Benjamin Werner, *The not so simple proof-irrelevant model of CC*, Second International Workshop on Types for Proofs and Programs, TYPES 2002, Berg en Dal, The Netherlands, April 24–28, 2002, Selected Papers, Lecture Notes in Computer Science, vol. 2646, Springer, 2003, pp. 240–258.

[NBS06] Tobias Nipkow, Gertrud Bauer, and Paula Schultz, *Flyspeck I: Tame graphs*, 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. 21–35.

[NG14] Rob Nederpelt and Herman Geuvers, *Type theory and formal proof*, Cambridge University Press, 2014.

[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.

[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.

[SU98] Morten Heine Sørensen and Pawel Urzyczyn, *Lectures on the curry-howard isomorphism*, Studies in Logic and the Foundations of Mathematics, vol. 149, Elsevier, 1998.

[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.

[TGMW10] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk, *Proviola: A tool for proof re-animation*, Proceedings of the Conferences on Intelligent Computer Mathematics, CICM 2010, (AICS 2010, Calculemus 2010, MKM 2010), Paris, France, July 5-10, 2010, Lecture Notes in Computer Science, vol. 6167, Springer, 2010, pp. 440–454.

[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.

[Wer94] Benjamin Werner, *Une théorie des constructions inductives*, Ph.D. thesis, Université Paris 7, May 1994.

[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.