CITP for CafeOBJ
ver. 1.2

澤田 寿実
(株) 考作舎
tswd@kosakusha.com

2016/2/29

目次

1

用語

本書は形式的証明について基礎的な知識を持つ CafeOBJ ユーザーを対象とした CITP for CafeOBJ(以降ではCITPと略称する)と呼ばれる証明譜作成支援システムの利用ガイドである． 本書で記述する CITP の機能は version 1.5.4 以降のCafeOBJ システムで利用可能である[1]．

表1.1に以下で使用するいくつかの用語の定義を示す．

[1]最新版の CafeOBJ システムは http://cafeobj.org/download/ からダウンロード可能である)

表 1.1: 用語の定義

用語	定義
文(sentence)	自由変数を含まない(条件付き)等式($t = t' if C$) あるいは、(条件付き)遷移規則 ($t \Rightarrow t'$)． CafeOBJ の等式宣言や遷移規則宣言のフォームで表記する．
文脈(context)	証明を実施するモジュール．CafeOBJ で提供されているコマンドには， ある特定のモジュールを対象としたコマンドが多数存在する． CITP for CafeOBJ で提供されるコマンドも多くがそうであり， それらのコマンドを使う際に，一々対象とするモジュールを指定するのは煩わしいために導入されたのが**文脈**という概念である． 文脈は “select” や “open” コマンドによって設定され， モジュールをパラメータとして持つコマンドの適用先を暗黙的に指定した–つまりパラメータを省略した際に適用先として採用されるモジュール–となる．
ゴール(goal)	四つ組 $< M, G, C, H >$． 文脈(M)とそこで証明したい文の集合(G)， 証明に際して使用した戦略(tactic)により導入された定数(constants)の集合(C)と 仮定(hypothesis)H(文の集合)．単に証明対象とする文(の集合)をゴールと呼ぶことがある．
戦略(tactic)	証明で用いる演繹規則およびそれらを組み合わせたものを戦略(tactic)と呼ぶ．
基底項(ground term)	変数を含まない項．

2

CITP for CafeOBJ のコマンド

本章では CITP for CafeOBJ で提供されるコマンドの挙動について述べる． 厳密な動作の理解のためには参考文献を参照されたい．

2.1 証明の開始(ゴールの設定)

ある特定のモジュール M を文脈として，その中で証明したい文の集合 G がすべて成立することを演繹規則(戦略)を順次適用することによって証明することが本システムの目標である．

- 証明を開始するには，証明を行う文脈を設定し，次いで証明したい文の集合を宣言する．
  文脈の設定は，既存の CafeOBJ コマンド **select <ModuleExpression> .** あるいは **open <ModuleExpression> .** によって行う．
- 証明を実施する文脈の設定後，下に示す **:goal** コマンドによって証明したい文の集合を指定する．

```
goal コマンド ::= :goal { <sentence> .  ...  <sentence> . }
```

sentence は，CafeOBJ の(条件付き)等式あるいは(条件付き)遷移規則のいずれかで表記する．

文脈とゴールの設定例

```
select CLOUD  .
:goal {
 ceq [inv1 :nonexec]: true = false if statusp(S:Sy s,I:Client) = updated /\
                                      statusc(S:Sy s)= idlec .
 ceq [inv2 :nonexec]: true = false if statusp(S:Sy s,I:Client) = gotval /\
                                     statusc(S:Sy s)= idlec .
 ceq [inv3 :nonexec]: true = false if statusp(S:Sy s,J:Client) = updated /\
                                     statusp(S:Sy s,I:Client) = gotval .
```

```
ceq [inv4 :nonexec]: true = false if (I:Client ~ J:Client) = false /\
                                    statusp(S:Sy s,J:Client) = gotval /\
                                    statusp(S:Sy s,I:Client) = gotval .
ceq [inv5 :nonexec]: true = false if (I:Client ~ J:Client) = false /\
     statusp(S:Sy s,J:Client)= updated /\ statusp(S:Sy s,I:Client)= updated .}
```

2.2 証明木の構造

ここでは他のコマンドを説明する前に，CITP for CafeOBJ の構成する 証明木(proof tree)の構造について説明する．

証明木は全体としてゴールをノードとする有向の木構造(directed tree structure)を持ち， 各枝(branch)はゴールに対して適用した戦略をラベル(label)が付加されている． あるゴールGに対してある戦略Tを適用すると， 一般的に複数のゴール$G_1, G_2, \ldots, G_n$ が生成されるが， これら G_i をもとのゴールの子ゴールと呼ぶ． G から各 $G_i(i = 1 \cdots n)$ への枝は，全て適用した戦略Tによるラベル Tを持つ．

以下この一般的な証明木の構造をより具体的に述べる． 以下では，ゴールと証明木のノードを区別せずに用いる．

- :goal コマンドによって設定されたゴールを初期ゴールとする．
- あるノード(ゴール)に対して戦略 T の適用によって新たなゴールが生成された場合， それらをそのノードの子ノードとする．これら子ノードへの枝はラベル T を持つ．
- 各ゴールには次のように名前が付加される:
  - 初期ゴールには root という名前が付けられる．
  - root 直下のゴールには自然数 $1, 2, \ldots, n$ と名前が付けられる．
  - 以下，あるゴールの名称が N であったとすると，そのゴールには N-1, N-2, ... N-m のように名前が付けられる．

このような構造を持つ証明木において，**ゴールが証明される**とは以下の事を言う:

- あるゴールの子ゴールが全て証明されたとき，そのゴールも証明される
- ここで**証明される**とは以下の事を言う
  1. 文が充足(satisfied)された場合，あるいは
  2. 矛盾(contradiction)が発見される: すなわち
     - その文脈において true = false が演繹可能(deducible)となる
     - 遷移的な関係で矛盾が生ずる．例えば $X < Y < Z$ の時に $Z \leq Y$ が演繹できる．

  これらのいずれかが成立した時，そのゴールに含まれる当該文はゴールから 取り去られる(dischage されると言う)． ゴールから証明対象の文がすべてなくなった時，そのゴールは証明されたと言う．

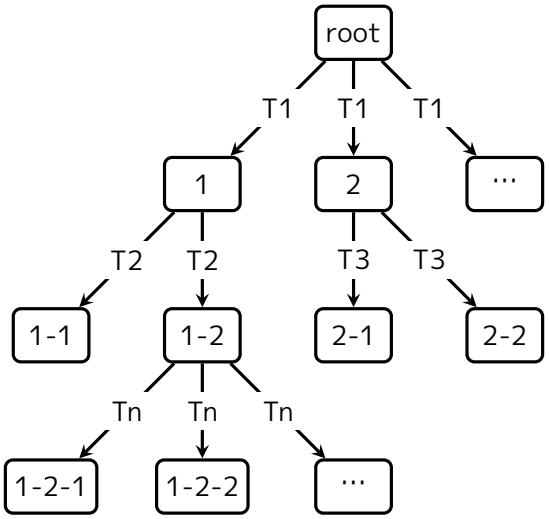

図 2.1: 証明木

表 2.1: 戦略一覧

戦略名	演繹規則
SI	Simultaneous Induction (同時帰納法)
CA	Case Analysis (場合分け)
TC	Theorem of Constants
IP, IP+	Implication (含意)
RD, RD-	Reduction (簡約化)

2.3 戦略の適用

- 初期ゴールが設定された後では，:apply コマンドによって，指定の戦略をゴールに適用できるようになる． ゴールが設定されていない場合，:apply コマンドの適用はエラーとして扱われる．
- 構文

```
apply  コマンド ::= :apply  [ to <GoalName> : ] (<Tactic> ... <Tactic> )
<GoalName>     ::= ゴールに付与された名前
<Tactic>       ::= SI | CA | TC | IP  | IP + | R D  | R D - | <D efinedTactic>
```

- **<Tactic>** の指定で，大文字と小文字の区別はしない．
- **to <GoalName>** が省略された場合は，現在のデフォルトゴール(後述)に対して適用される．
- **<D efinedTactic>** は2.8.11章で説明する **:def** コマンドによって定義された戦略の名前である．

2.3.1 戦略

CITP for CafeOBJ で提供される戦略を表 2.3.1 に示す． 各戦略の具体的な挙動については，2.5章で説明する．

2.3.2　戦略の適用順序

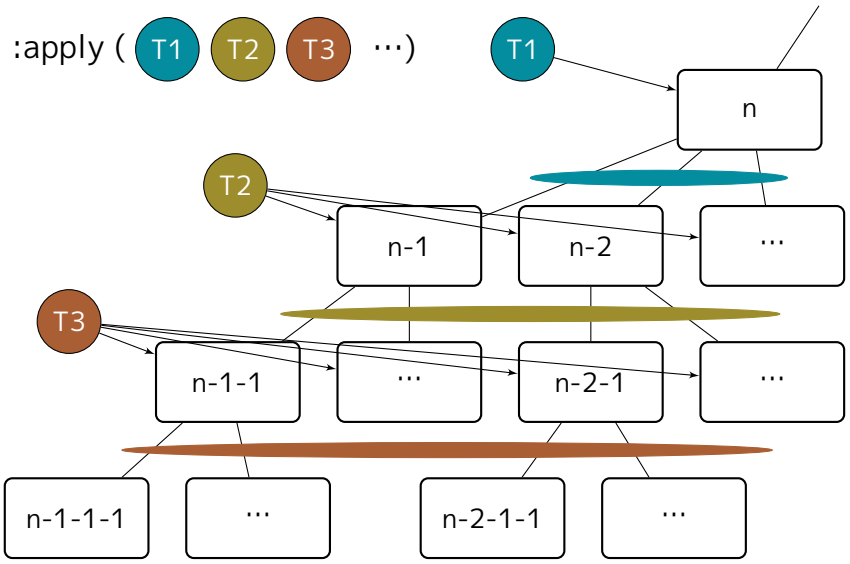

図 2.2: 戦略の適用順序

図 2.2 に，あるゴール n に対して一連の戦略 $T_1T_2 \ldots T_n$ を 適用した際に，これらの戦略がどのように適用されるかを示す． 一般にある戦略 T_i をゴール N に適用した場合，戦略によって複数のゴールが生成される． :apply コマンドに一連の戦略 $T_1T_2 \ldots T_n$ が指定され，ゴール N に適用されたとする． このとき最初の戦略 T_1 によって複数の小ゴール $N_1 \ldots N_m$ が生成されたとした時， 次の戦略 T_2 はこれらの小ゴール全てに対して適用される．以下同様である．

2.3.3　自動戦略

経験則から一般的に有効(うまく証明ができることが多い)と考える事のできる一連の戦略を予め用意し， これを簡便に使用できると便利である． このためにコマンド :auto が用意されている．

- 構文:

```
auto コマンド ::= :auto
```

- **:auto** は **:apply　(SI CA TC IP　R D)** と等価である．

2.4　ターゲット・ゴール

:apply コマンドは戦略を適用するゴールの引数を省略することができる(2.3)． この時，指定した戦略の適用対象となるゴールを**ターゲット・ゴール**と呼ぶ． システムはこれを次の規則によって決定する．

- 初期のゴールが **:goal** コマンドによって設定された直後は **root** がターゲット・ゴールである．

・ある戦略をゴールに適用した後は，証明木の構造上最も左の 末端ノードがターゲット・ゴールとなる．

ターゲット・ゴールは :apply コマンドの暗黙的な対象となるゴールだけでなく， ゴールを引数に持つコマンドで，それが省略された際に適用対象とするゴールとして扱われる．

2.4.1 ターゲット・ゴールの指定

ターゲット・ゴールは，**:select** コマンドによって設定することが可能である．

・構文:

```
:select コマンド ::= :select <GoalName>
```

:select コマンドで指定されたゴールが子ゴールを持っていた場合， それらの子ゴールは証明木から削除される． 図2.3 はゴール1-1がターゲット・ゴールとなっている状態で **:select** コマンドでゴー

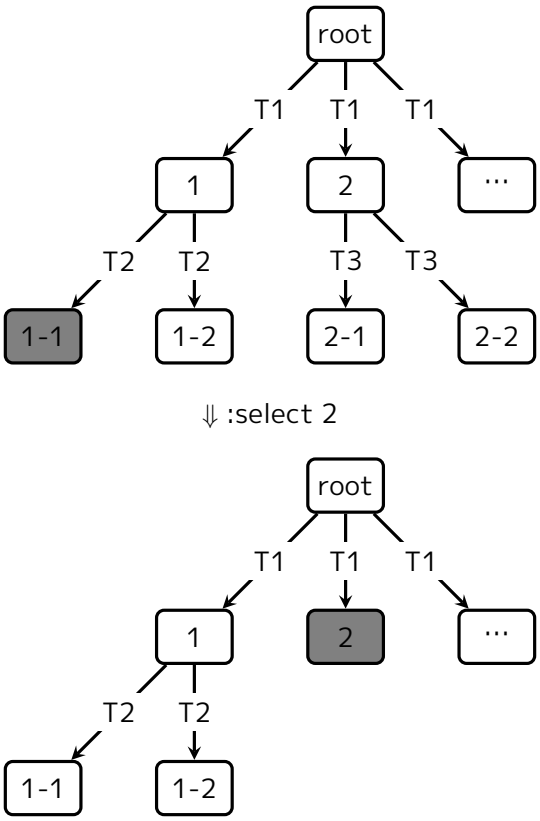

図 2.3: :select

ル2を指定した結果ゴール2がターゲット・ゴールとなり それまでゴール2の子ゴールだった既存のゴール2-1と2-2が削除された様子を示したものである．

2.5 各戦略の挙動

本章では先に述べた各戦略の挙動について述べる.

2.5.1 SI: Simultaneous Induction

$$[\text{conAbst}]\ \frac{\{\mathsf{SP} \vdash^{sp} (\forall Y)\theta(\varepsilon) \mid \theta : X \to T_{\Sigma^c}, Y : \mathbf{finite}\}}{\mathsf{SP} \vdash^{sp} (\forall X)\varepsilon}$$

ここで $\Sigma = \mathbb{S}\mathrm{ig}(\mathsf{SP})$, $\Sigma^c \subset \Sigma$ は構成子からなる部分シグニチャである. また $\theta : X \to T_{\Sigma^c}(Y)$ は $\mathbb{S}\mathrm{ig}(\mathsf{SP})$-置換である. 上の規則はもし ε が X に含まれる変数の全ての可能なinstantiationについて成り立つなら $(\forall X)\varepsilon$ が成り立つことを意味する.

[conAbst] は実用的な観点からは使用するのが難しい. 何故ならルールのインスタンスが無限にあり得るからである. その代わりに下の帰納的演繹によってこれを模倣する事が出来る.

$$\mathsf{SP}' =_{def} \mathbf{P\ R}\ (\mathsf{SP}, \{\{x\}_s\}) \cup \{(\forall\{\})\varepsilon\}$$
$$[\text{Ind}]\ \frac{\mathsf{SP}' \vdash (\forall Z^f)\varepsilon[x \leftarrow f(z_1, \ldots, z_{i-1}, x, z_{i+1}, \ldots z_n)]\ \ f \in F^c_{*s}\}}{\mathsf{SP} \vdash^{sp} (\forall\{\{x\}_s\})\varepsilon}$$

Ind は構成子ベースの帰納的演繹を定式化したものである.

- 戦略 SI は上の Ind を複数の帰納変数に対して同時に適用可能なよう拡張したものである. 指定した帰納変数(induction variables)に対して以下を行う.
  - ベースケース
  - 帰納法の仮定+ステップケース

  それぞれに対応した新たなゴールを生成する
- induction variables は **:ind on** あるいは **:ind+ on** コマンドによって指定する.

```
:ind コマンド ::= :ind on ( <変数> ... )
:ind+ コマンド::= :ind+ on ( <変数> ...)
                  with base ( <項> . ...<項m> .)
                       step ( <項> . ...<項n> .)
```

  <変数> は on-the-fly の変数宣言形式で指定する.
- **:ind** あるいは **:int+** コマンドで指定した変数に対応するソートが, 構成子を持たないソートであった場合はエラーとする.
- 帰納法の仮定や, 証明対象とするステップケースの文では, 帰納法による定数が必要となるが, それらは次のようにして生成する.
  - **帰納変数名#ソート名** という名前のオペレータを導入
  - このオペレータを用いて(定数)項を生成する

  例えば, 帰納変数として **I:Foo** が指定された場合, 導入される定数名は **I#<ソート名>** となる. <ソート名> は文脈に応じて適切なソートが選択される.

利用者指定の帰納スキーム

:ind コマンドの場合，システムが自動的に指定の帰納変数のソートの 構成子定義からベースケースとステップケースに対応する項のパターンを 生成するが， **:ind+** は帰納法のベースおよびステップのパターンを， それぞれ **base** および **step** で指定するものである． **<項>** のパターンを引数として与え，それに基づいて各々の ケースの項のパターンを生成する．

2.5.2 TC: Theorem of Constants

$$[\text{TC}]\ \frac{\textbf{P R}\ (\text{SP}, Y) \vdash^{sp} (\forall\{\})\varepsilon}{\text{SP} \vdash^{sp} (\forall Y)\varepsilon}$$

この演繹規則は全称的に束縛された変数に関する推論を行う際に， 定数を代わって使用しても良いとするものであり， 従来よりCafeOBJの書き換えエンジンを利用した証明で通常利用されている技法である．

CITP CafeOBJ ではこの演繹規則に対応するものを戦略として提供する． [TC]によって新たに導入する定数名は，既存の定数名と衝突が無いように考慮し， 規則的に命名される．

- 適用先のゴールに複数の証明対象の文が含まれている場合は， それ毎に別々の小ゴールを作成し，分配する．
- 以下各ゴールに対して以下を実施する．
  - 証明対象の文に含まれる変数を対応するソートの定数で置き換える
- 定数は次のように新たなオペレータを導入することによって作成する
  - **変数名@ソート名** という名前のオペレータを導入
  - このオペレータを用いて(定数)項を生成する

  例えば，変数 X がソート Foo の変数であった場合，導入されるオペレータは

  ```
  op X@Foo : -> Foo .
  ```

  のように宣言されたのと等価である．

2.5.3 IP: Implication

$$[\text{IP}]\ \frac{(\Sigma, E \cup \{(\forall\{\})t_1 = t'_1, \ldots, (\forall\{\})t_n = t'_n) \vdash^{sp} (\forall\{\}t = t')}{(\Sigma, E) \vdash^{sp} (\forall\{\})t = t'\ \textbf{if}\ \{t_1 = t'_1, \ldots, t_n = t'_n\}}$$

条件付き等式で条件部が基底項の場合，それらを新たな公理として導入し， 条件部を取り去ったゴールを新たなゴールとしても良い事を [IP] は示している．

[IP] についても CITP for CafeOBJ の提供する演繹規則として提供する． 条件部が複数の atom の連結 ($\wedge$ で結ばれた複数の条件)の場合は， それらを別々の複数の公理として導入する．

- 適用先のゴールに複数の証明対象の文が含まれている場合は， それ毎に別々の小ゴールを作成し，分配する．

- 以下，各ゴールに対して以下を実施する．
- 証明対象の文が **ceq T = T' if C** または **ctrans T => T' if C** の形，かつ
- C が基底項の場合に以下を行う
  - **C** を公理として追加
  - 元の証明対象から条件部を削除したものを新たな証明対象の文とする
  - **C** が複数の条件が **c1** /\ **c2 ...** /\ **cn** のように，/\ で 結合された形の場合は分離し，個々の **c1** . . . **cn** を 公理として導入する．

2.5.4 IP+: Implication 2

IP+ の挙動は IP と同様であるが，条件部から生成された前提 C を公理として追加するのではなく，証明対象の文の左辺 LHS を，

```
C implies LHS = R HS
```

のように，組み込みのオペレータ implies を用いて導入する．

2.5.5 CA: Case Analysis

$$[\text{split}]\ \frac{\{\textbf{P R}\ (\mathsf{SP}, Y) \cup \{u = t\} \vdash^{sp} e \mid t \in T_{\Sigma^c}(Y)_{S_c}, Y : \textbf{finite}\}}{\mathsf{SP} \vdash^{sp} e}$$

$$[\text{splitBool}]\ \frac{\mathsf{SP} \cup \{u = \textbf{true}\} \vdash^{sp} e \qquad \mathsf{SP} \cup \{u = \textbf{false}\} \vdash^{sp} e}{\mathsf{SP} \vdash^{sp} e}$$

これらは網羅的な場合分けの必要性について定式化したものである． これについては証明の文脈となっている仕様や証明の対象としている項により異なり，一般的な生成スキームを与える事は困難である．

そのため，場合分けを実施する上での各ケースを利用者が公理によって明示的に指示し， システムがそれをベースに必要な場合分けを実施するものとする． この仕様は Maude の CITP システムに習ったものである．

CA は次のように動作する:

1. 証明対象の文から基底項 $G_1, \ldots, G_n$ を取り出す．
2. 文脈となっているモジュールから，ラベルとして先頭が “CA” で始まる 公理の集合 Ac を求める．
3. 各 $G_i (1 \leq i \leq n)$ について，その真部分項が Ac に含まれる公理の どの左辺とも照合しない基底項の集合 Gs を得る．
4. 各 $g_j \in Gs$ について，各 ($\textbf{l} = \textbf{r if C}$) $\in Ac$ との間で 以下を計算する．
   a) 基底項 g_j に関するケースの集合 C_j を空集合にセットする．

b) $\sigma(g_j) = \mathtt{l}$ となる置換 σ が存在したら， $\sigma(C)_j$ を Cs_j に追加する．

各 Cs_j は基底項 g_j に関するケースの集合となっているので， これらの全ての組み合わせ $CS_1 \times S_2 \times \cdots \S_m$ を計算し， 全ての可能なケースの組み合わせを求める．

5. 上で得られた各組み合わせごとに，新たな子ゴールを生成し， そのゴールへケースを公理として追加する．

冗長なケースの生成があり得るための，それらを検査して取り除く事は 実装に際して最適化の意味で実施するものとする．

- 上で述べた CA 処理を実施するする前に，適用対象のゴールに複数の 証明対象の文が含まれていたら，それらを個々の小ゴールに分配し， その後，上の CA 処理を実施する．

2.5.6 RD: Reduction

戦略 RD は，以下を実施する：

1. ゴール内で true = false が演繹可能かどうかを調べる．

$$[\mathrm{CT}]\ \frac{\mathrm{SP} \vdash \mathbf{true} \Rightarrow \mathbf{false}}{\mathrm{SP} \vdash \rho}$$

これが可能な場合，矛盾なのでゴールに含まれるすべての証明対象としている文を discharge する．

2. ゴールに含まれる証明対象の文すべてについて以下を行う． ここで，証明対象の文を `l = r if C` とする．
   a) l，r あるいは C のどれかが基底項でなければなにもしない． これらのすべてが基底項の場合にのみ以下を実施する．
   b) 条件部 `C` の既約形(normal form)を求める． 結果が `true` ならば次へ進む(条件部が存在しない場合は `true` とみなされる）． 結果が `true` でなければ文はまだ成立しないとみなす．
   c) 文の左辺 `l` の既約形を求める．
   d) 文の右辺 `r` の既約形を求める．
   e) 左右両辺の既約形が「等しい」かどうかを調べる． ここで，等しいとは以下のことを言う：
      - 項の形が同じ．
      - 項のトップオペレータがセオリー属性(associative, commutative など)を持っていた場合， その意味で等しい．
   f) 等しければ成立するとし，その文を discharge する．等しく無ければ成立しない，とする．
3. ゴールに含まれるすべての文が discharge された場合，そのゴールは証明されたものとする．

他の戦略は証明できるかどうかを直接調べないが，戦略 RD はゴールが証明できるかどうかを上のようにして調べる．

2.5.7 RD-

戦略 RD- の挙動は RD と似ているが，以下の点が異なる．

RDでは証明対象の文の左右辺の既約系を求める際に，これらを項書き換えによって 破壊的に変形する．そのため文が成立しなかった場合，それらの文は既約形に 書き換えられた形となる．それに対してRD-では証明対象の文が成立するかどうかを 調べる際に，もとの文を破壊的に書き換えることなく調べる[1]．

2.6 :spoiler：暗黙的 RD 適用の制御

2.5 でシステムに組み込みの各戦略の動作を述べた． RD 以外の戦略はその戦略に応じた新たな仮定を導入したりするなどするが， その結果ゴールに含まれる文が成立するかどうかは調べない． システムは **:spoiler** という名称のフラグを持っており， これを on に設定することにより，戦略適用後の状態でゴールが証明可能かを調べるよう指示することができる． ただし戦略 SI はこの対象外であり， このフラグが on の場合でも2.5.1で述べた以上の処理は行わない．

このフラグの設定は **:spoiler** コマンドによって行う．

```
:spoiler コマンド ::= :spoiler { on | off }
```

フラグの初期値は off である．

このフラグが on の場合の各戦略(SIおよびRDを除く)の挙動は，2.5 で説明した処理を実施後に戦略 RD を実施するのと等価である．

:auto の動作

:auto コマンドによる戦略適用の場合は， :spoiler フラグが on の状態で実行される． :auto コマンドの実行が終了後はフラグは元の状態となる．

2.7 補助的戦略

ここでは演繹規則として位置付けられるものではないが， 証明過程で有用と考えられる補助的な戦略について説明する．

2.7.1 NF:証明対象文の既約形

- ある戦略を適用した後にゴールに含まれる証明対象の文に 含まれる基底項(ground term)を全て既約形(normal form)にする．

[1] 左右辺の既約形を求めて調べる点は同じである． 一旦コピーを作り，それらを書き換えて調べるのと等価な動作を実行する．

NF は **:apply** コマンドの引数として与えることができる.

2.8 その他のコマンド

2.8.1 :init コマンド

- **:init** コマンドは証明の途中で lemma の導入と初期化を行うためのコマンドである.
- 構文

```
:init コマンド ::= :init "["<label>"]" by  <Substitution>
                |  :init "(" <axiom> ")" by  <Substitution>
Substitution ::= "{" <Variable> <- <Term> ; ... <Variable> <- <Term> ; "}"
```

- これを実行することによって，**<label>** で指定されたラベルを持つ公理， あるいは "(" と ")" で囲まれた公理に含まれる変数を **Substitution** で 示された変数置換によって初期化した公理を，ターゲット・ゴールの公理として追加する.
- 追加する公理は，後で見た時に :init コマンドによって導入された事が分かるよう [INIT] というラベルを付加する.

:init コマンドの例

文脈となっているモジュールに，次のような公理があったとする.

```
ceq[inv3 :nonexec]: true = false if statusp(S:Sy s,J:Client) = updated /\
                                    statusp(S:Sy s,I:Client) = gotval .
```

これに対して下のような :init コマンドを適用できる.

```
:init [inv3] by  {S:Sy s <- S#Sy s ; J:Client <- I@Client ; I:Client <- S#Client ;}
```

これを実行することによって，新たな公理

```
ceq [INIT]: true = false if statusp(S#Sy s, I@Client) = updated /\
                            statusp(S#Sy s, S#Client) = gotval .
```

が追加される.

2.8.2 :imply コマンド

- :imply コマンドは :init コマンドと似ているが，指定可能な等式は実行文脈に すでに存在している **eq[label]: lhs = true .** という形の等式しか指定することができない.
- 構文

```
:imply  コマンド ::= :imply  "["<label>"]" [ by  <Substitution> ]
Substitution   ::= "{" <Variable> <- <Term> ; ... <Variable> <- <Term> ; "}"
```

- 挙動

1. **<label>** で指定されたラベルを持つ等式に含まれる ‘eq [label]: lhs = true. ’ という形の等式の左辺 **lhs**に**<Substitution>** で指定された変数置換 σ を適用する.
2. 得られた σ(lhs) の既約形 norm(σ(lhs))を求める.
3. 証明対象の文が ’eq L = R .’ という形のものだとすると，これを ’eq norm(σ(lhs)) implies L = R .’ という形の文で置き換える.

変数置換(**<Substitution>**)は省略可能である．この場合，証明対象の文は ’eq lhs implies L = R . ’ という形の文で置き換えられる.

2.8.3 :roll back コマンド

- :roll back は，現在のターゲット・ゴールに対して適用された，直前の戦略を キャンセルする.
- 構文

```
:roll back コマンド ::= :roll back
```

- このコマンドの実行により，ターゲット・ゴールは証明木から削除される.

2.8.4 :cp コマンド

- :cp コマンドは指定した2つの文のクリティカルペアを求め， 利用者に提示する.
- 利用者はそれに対して，次節で述べる :equation コマンド等を用いて， それを公理としてターゲット・ゴールへ追加する事ができる.
- 構文

```
:cpコマンド ::= :cp <Sentence> >< <Sentence>
<Sentence>  ::= "["<Label>"]" | "(" <axiom> . ")"
```

- **<Sentence>** は，文脈モジュールで宣言されている公理のラベルを **<Label>** で指定するか，あるいは CafeOBJ の公理宣言フォームを "("と")"で囲んで記載する.

:cp コマンドの使用例

この例は，直接文を CafeOBJ の公理の宣言記法で記述しクリティカル・ペアを 求めている例である.

```
:cp (ceq top(sq(S@Sy s)) = I@P id if pc(S@Sy s,I@P id) = cs .)
   ><
   (ceq top(sq(S@Sy s)) = J@P id if pc(S@Sy s,J@P id) = cs .)
```

2.8.5 :equation/:rule コマンド

- これらのコマンドは :cp コマンドで得られたシステムからのクリティカル・ペアの 提示に対する，利用者の回答として使用される.
- 構文

```
:cp回答 ::= :euqation | :rule
```

- **:equation** はクリティカル・ペアを等式としてターゲット・ゴールへ追加する.
- **:rule** はクリティカル・ペアを遷移規則としてターゲット・ゴールへ追加する.

:equation コマンドの使用例

下は :cp コマンドによるクリティカル・ペアを等式としてターゲット・ゴールへ追加する例である.

```
QLOCK(X) > :cp (eq I@P id = S#P id .) >< (eq S#P id ~ I@P id = false .)
[cp] :
  (1) (true):B ool
      => (false):B ool
QLOCK(X)> :equation
[cp] added cp equation to goal "4-1-1-1":
  eq [CP ]: true = false
[ip]=>
:goal { ** 4-1-1-1 ----------------------------------------
  -- context module: QLOCK
  -- induction variable
    S:Sy s
  -- introduced constant
    op I@P id : -> P id { prec: 0 }
  -- constants for induction
    op S#Sy s : -> Sy s { prec: 0 }
    op S#P id : -> P id { prec: 0 }
  -- introduced axioms
    ceq [SI :noexec]: top(sq(S#Sy s)) = I:P id if pc(S#Sy s, I:P id) = cs .
    ceq [INIT]: top(sq(S#Sy s)) = I@P id if pc(S#Sy s, I@P id) = cs .
    eq [CA]: pc(S#Sy s, S#P id) = cs .
    eq [CA]: S#P id ~ I@P id = false .
    ceq [INIT]: top(sq(S#Sy s)) = I@P id  if pc(S#Sy s, I@P id) = cs .
    eq [IP ]: pc(S#Sy s, I@P id) = cs .
    eq [CP ]: true = false .
  -- axiom to be proved
    eq [TC :noexec]: top(get(sq(S#Sy s))) = I@P id .
}
```

追加される公理は，:cp コマンドの結果追加された事が後で判別できるよう， ラベルに CP が付けられる.

2.8.6 :backward equation/rule コマンド

先の節で述べた :equation および rule コマンドと同様だが， 提示されたクリティカル・ペアの右辺と左辺を入れ替えた公理として ターゲット・ゴールへ導入する．

2.8.7 :ctf コマンド

:ctf コマンドは，指定した等式や遷移規則規則の成立・不成立， あるいは指定した項が定数構成子に等しいか否かによる場合分けを行う．

等式／遷移規則による場合分け

:ctf コマンドの下に示す構文はある等式あるいは遷移規則が成立する場合と成立しない場合の2ケースで， 現在のターゲット・ゴールを2つのサブゴールに分割する．

- 構文1：等式あるいは遷移規則の成立／不成立による場合分け

```
true/falseによる場合分け ::= :ctf "{" { <Equation> . | <Transition> . } "}"
```

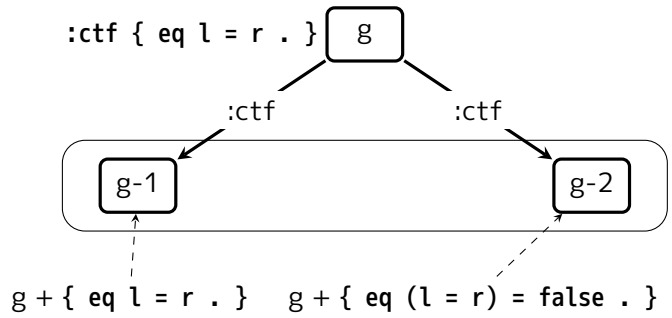

図 2.4: :ctf の動作 – 等式／遷移規則の成立・不成立による場合分け

図2.4 はゴール g がターゲット・ゴールの状態で

```
:ctf {eq l = r . }
```

とした時の場合分けの様子を示したものである． ゴール g に **eq l = r .** を仮定として追加した g-1 (true の場合)と， **eq (l = r) = false .** を追加した g-2 (false の場合)を作成し， ゴール g の子ゴールとする．

構成子による場合分け

:ctf コマンドは構成子を持つソートの項を指定し， その構成子による場合分けを行うこともできる．その場合の構文は下のとおりである．

- 構文2：定数構成子による場合分け

```
構成子による場合分け ::= :ctf "[" <項> . "]"
```

図2.5 はゴール g がターゲット・ゴールの状態で **:ctf [t .]** として指定した項 **t** が 組み込みのソート**B ool** の項である場合の例を示したものである． **t** が **true** の場合と **false** の場合に場合分けされている．

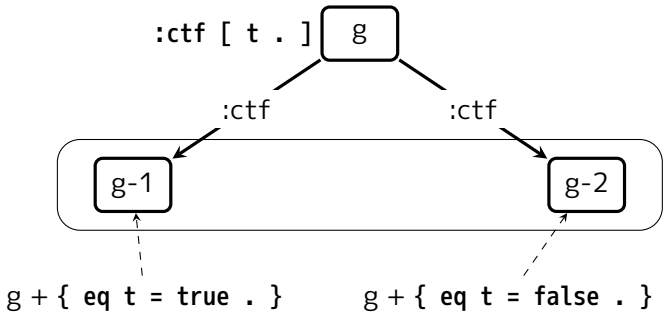

図 2.5: :ctf の動作 – 定数構成子による場合分け

:spoiler on の場合の挙動

:spoiler フラグが on の場合，このようなサブゴールに分割後，それぞれのサブゴールで暗黙的に戦略 **R D** (2.5.6) を実行する．

2.8.8 :ctf- コマンド

:ctf- は **:ctf** コマンドと場合わけの機能については全く同じである． しかし作成された子ゴールに対して **R D** を適用した結果， 証明対象の文が discharge **されなかった**場合に次のような違いがある：

- **:ctf** コマンドの場合
  証明対象の文は **R D** の適用によって簡約化された状態のままとなる
- **:ctf-** コマンドの場合
  証明対象の文は **R D** が適用される前の状態に戻される

一般に **R D** の適用により証明対象の文は破壊的に書き換えられるため， 場合分けを連続的に実施していくような証明の過程で **R D** 戦略を 適用した場合[2]， うまく discharge できなかった際は元の証明対象の文に戻して さらに場合分けを続けていきたい． これを自動的に行うのに **:ctf-** が便利である．

2.8.9 :csp コマンド

:csp コマンドは複数の等式または遷移規則を指定し， 各々が成立するとした小ゴールを作成する．もし :spoiler フラグが on の場合は， その後，各小ゴールに対して暗黙的に戦略 RD (2.5.6) を適用する．

[2] :spoiler フラグが on の場合は常にそうなる

・ 構文

```
ケース指定分割 ::= :csp "{" { <Equation> . | <Transition> .}+ "}"
```

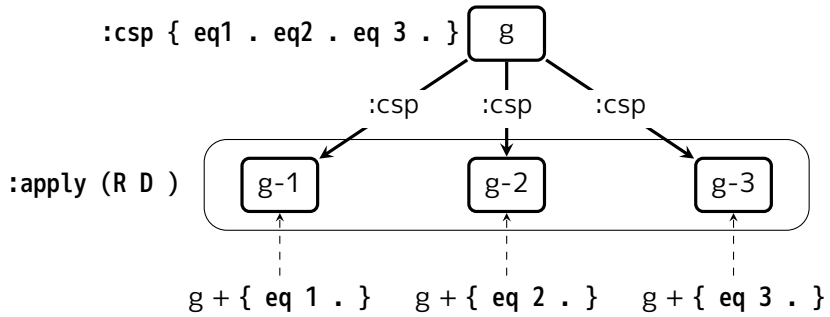

図 2.6: :csp の動作

図 2.6の説明　ターゲットゴールが g だとする．":csp eq1 . eq2 . eq3 . " のようにして 3つの等式 eq1, eq2, および eq3 を指定した場合，:csp は3個の小ゴール g-1, g-2, g-3 を生成し， それぞれに対して一つづつ指定された等式を配分する． 図は :spoiler フラグが on の状態を示しており， 生成された小ゴールに対して暗黙的に戦略 RD を適用する様子も示している．

2.8.10　:csp- コマンド

:csp- コマンドは場合わけの機能については **:csp** コマンドと全く同じである． 構文も同様ででキーワード **:csp** に代わって **:csp-** を指定する． 違いは **:ctf** と **:ctf-** の違いと同じである． すなわち，生成された子ゴールに対して適用された **R D** 戦略の結果 discharge されなかった場合に，元の証明対象の文に戻す(**:csp-**)か 戻さない(**:csp**)かの違いである．

2.8.11　:def コマンド

連続した戦略の適用に名前をつける

:def コマンドは一連した戦略適用に名前をつけ **:apply** コマンドの 引数として与えることができるようにする．

```
一連の戦略定義 ::= :def <name> = ( <戦略1> ... <戦略n> )
```

下は戦略 **IP** に引き続いて **R D** を適用することに **ip+rd** という名前をつけたものである．

```
:def ip+rd = ( IP  R D  )
```

上のようにすることにより，**:apply** コマンドの引数として **ip+rd** を 指定することができるようになる．

```
:apply  (... ip+rd ...)
```

上の **ip+rd** の定義は :spoiler フラグが on の場合の戦略 **IP** の 挙動と同じである．このように :spoiler フラグが off の状態(デフォルト) で **R D** を自動的に実行したい場合はこのような定義をしておくことで 簡便にこれを行うことができる．

:ctf などの戦略的な使用

また，**:ctf** コマンドの指定や **:csp** コマンドの指定に名前をつけ， **:apply** コマンドの引数として与えることができるようにする．

```
:ctf 戦略定義 ::= :def <name> = :ctf "{" { <Equation> . | <Transition> . } "}"
              |   :def <name> = :ctf [ <Term> . ]
:csp 戦略定義 ::= :def <name> = :csp "{" { <Equation> . | <Transition> .}+ "}"
```

:ctf- や **:csp-** も同じである．

このようにすることで，**:ctf-** などのコマンドの適用を，他の戦略と組み合わせて **:apply** コマンドの引数として与えることができる．

2.8.12 :show コマンド

証明の進行状況などを確認するために有用と思われる情報をみるため， 既存の show コマンドと類似の機能を提供する **:show** コマンドを提供する．

ゴール内容の表示 – :show goal

- ゴールの内容を表示する．
- 構文

```
    ゴールの表示 ::= :show goal [ <GoalName> ]
```

- **<GoalName>** で指定したゴールを表示する．
- **<GoalName>** が省略された場合は，現在のターゲット・ゴールを表示する．

未証明のゴールの表示 – :show unproved

- 現時点でまだ証明されていないゴールを表示する．
- 未証明のゴールとは，現在の証明過程における証明木で，末端のノードうち まだ証明対象の文が discharge されていないゴールの事を言う．
- 構文

```
      未証明のゴール表示 ::= :show unproved
```

証明木の構造表示 – :show/:describe proof

:show proof

- 現時点における証明木の構造を図式的に表示する.
- 構文

```
証明木の表示 ::= :show proof
```

- 証明木の表示にあったっては，以下の事が容易に判別出来るようにする.
  - ゴールがどの戦略によって生成されたものであるか
  - ターゲット・ゴールが何であるか
  - ゴールは証明済みか否か

:describe proof

- **show proof** と同様だが図式的な構造の表示では無く，証明の過程で 使用された演繹に関する情報を提示するのが目的である.
- 構文

```
証明過程の表示 ::= :describe proof
```

- 表示にあたっては，以下の情報を提示する.
  - そのゴールを生成した戦略
  - 証明すべき文
  - 帰納法の対象とした変数
  - SI あるいは TC 戦略を適用するにあたって導入された定数
  - 戦略によって導入された公理
  - ゴールが証明されているか否か

3

例題

本章では 2で説明した CITP for CafeOBJ の基本的な機能を網羅した例題について， その実行例と合わせて説明する．

3.1 足し算の性質の帰納法による証明

ペアノ流の自然数の上で足し算を定義し，足し算の交換則と結合則が成り立つことを証明する．

PNATの定義

自然数のソートは **P Nat** とし，下位ソートとして **P Zero** (ゼロ)と **P NzNat** (1以上の自然数)を定義する．足し算は **+**， **0** と **s** が構成子である．

```
**
** P rove associativity  and commutativity  of addition
** using CITP  for CafeOB J
**

mod! P NAT {
  [ P Zero P NzNat < P Nat ]
  op 0 : -> P Zero {ctor} .
  op s_ : P Nat -> P NzNat {ctor} .
  op _+_ : P Nat P Nat -> P Nat .
  eq 0 + N:P Nat = N .
  eq s M:P Nat + N:P Nat = s(M + N) .
}
```

3.1.1 準備

結合則の証明に必要な性質として次の性質がある．

```
N:P Nat + 0 = N:P Nat  .
M:P Nat + s N:P Nat = s(M:P Nat + N:P Nat) .
```

そのため，まず最初に足し算+で，これらの性質を先に証明する． CIT for Cafe で証明するため，最初に PNAT を文脈として設定してから 証明したいゴールを設定する．

```
select P NAT .
:goal { eq [lemma-1]: M:P Nat + 0 = M:P Nat .
        eq [lemma-2]: M:P Nat + s N:P Nat = s(M:P Nat + N:P Nat). }
```

上の **:goal** コマンドの実行結果は次のようになる．

```
:goal { ** root ---------------------------------------
  -- context module: P NAT
  -- axioms to be proved
    eq [lemma-1]: M:P Nat + 0 = M .
    eq [lemma-2]: M:P Nat + s N:P Nat = s (M + N) .
}
** Initial goal (root) is generated. **
```

ゴール名が **root** であり，先に **:goal** コマンドで指定した文が 証明対象となっている事が示されている．

証明は変数 **M:P Nat** の上の帰納法によって行う．そのため **:ind on** コマンドで 帰納法で使用する変数を宣言する．証明の過程で生成されるゴールを確認したいため **:verbose** コマンドで **on** を指定する． デフォルトでは **:verbose** の値は **off** である．

```
:ind on (M:P Nat)

**> We want to see every  goal generated in proof process.
:verbose on
```

上を実行するとシステムは M の上で帰納法を用いた証明を実施する旨表示する．

```
    **> Induction will be conducted on M:P Nat
```

証明は戦略 **:auto** で行う．これは戦略 **(SI CA TC IP R D)** と等価である(2.3.3)． また，**:auto** の実行中は，:spoiler フラグが on の状態で戦略が適用される(2.6)．

```
:auto
```

以下順次，**:auto** コマンドの出力を示す．

```
[si]=> :goal{root}
** Generated 2 goals
[si]=>
:goal { ** 1 ----------------------------------------
  -- context module: P NAT
  -- induction variable
    M:P Nat
  -- sentences to be proved
    eq [lemma-1]: 0 + 0 = 0 .
    eq [lemma-2]: 0 + s N:P Nat = s (0 + N) .
}
[si]=>
:goal { ** 2 ----------------------------------------
  -- context module: P NAT
  -- induction variable
    M:P Nat
  -- constant for induction
    op M#P Nat : -> P Nat { prec: 0 }
  -- introduced axioms
    eq [SI lemma-1]: M#P Nat + 0 = M#P Nat .
    eq [SI lemma-2]: M#P Nat + s N:P Nat = s (M#P Nat + N) .
  -- sentences to be proved
    eq [lemma-1]: s M#P Nat + 0 = s M#P Nat .
    eq [lemma-2]: s M#P Nat + s N:P Nat = s (s M#P Nat + N) .
}
```

まず最初に戦略 **si** が適用され，2つのゴールが生成されている． ゴール 1 は帰納法のベースケースであり，もとの証明対象の文にあった 帰納法の変数 M を構成子 0 とした文が証明対象となる．

ゴール 2 はステップケースである．帰納法の仮定として，“ある自然数 N で 成り立つとしたとき”に相当する文が導入されている．ある自然数 N に相当する 項のために定数 **M#P Nat** が使われている．この定数を定義するために 導入されたオペレータ宣言についてもゴールの表示で示されている．

証明対象とする文は，上で述べた仮定が成り立つとした時に示すべき文である． そのため構成子 **s** により **M#P Nat** が展開されている．

戦略 si を適用したあと，システムは ca を自動的に適用する． ca はまず最初にゴール1に対して適用される． 複数の証明対象がゴールに含まれているため， それぞれを別々の子ゴールに分配し，それぞれでケース分けが 可能かどうかを調べる． この場合ケースわけは実行されなかった(ケースわけの対象とする 公理は宣言されていない)．

ca はケースわけ分析が終了した後に，:spoiler フラグが on の場合に限って 暗黙的に戦略RDに相当する処理(文の充足性と矛盾の有無の調査)を行うが， その結果一つの文が充足されておりその旨出力

する． 具体的な出力は次のようになる．

```
[ca]=> :goal{1}
[ca] discharged: eq [lemma-1]: 0 = 0
** Generated 2 goals
[ca]=>
:goal { ** 1-1 ----------------------------------------
  -- context module: P NAT
  -- discharged sentence
    eq [ST lemma-1]: 0 = 0 .
  -- induction variable
    M:P Nat
} << proved >>
[ca]=>
:goal { ** 1-2 ----------------------------------------
  -- context module: P NAT
  -- induction variable
    M:P Nat
  -- sentence to be proved
    eq [lemma-2]: 0 + s N:P Nat = s (0 + N) .
}
```

:auto は一連の戦略適用 (SI CA TC IP RD) に等しいことを先に述べたが、 戦略CAで生成されたゴール 1-1 および 1-2 に対してCA以降の戦略が 順に適用されて行く。1-1 は既に今回のCAの適用によって discharge されているため、各戦略は何もしない。 1-2 はまだ証明すべき文が残っているため、戦略適用の対象となる。

```
[tc]=> :goal{1-1}
[ip]=> :goal{1-1}
[rd]=> :goal{1-1}
[tc]=> :goal{1-2}
[tc] discharged:
  eq [TC lemma-2]: s N@P Nat = s N@P Nat
[tc] discharged the goal "1-2-1"
** Generated 1 goal
[tc]=>
:goal { ** 1-2-1 ----------------------------------------
  -- context module: P NAT
  -- discharged sentence
    eq [TC TC lemma-2]: s N@P Nat = s N@P Nat .
  -- induction variable
    M:P Nat
  -- introduced constant
    op N@P Nat : -> P Nat { prec: 0 }
} << proved >>
[ip]=> :goal{1-2-1}
[rd]=> :goal{1-2-1}
```

TCは証明対象の文の変数を定数項で置き換えた後， もし :spoiler フラグが on であれば文が成立するか否かを調べる． 現在は :auto で実行しており、従って :spoiler が on の状態であるため, 上の実行の様子のとおりゴール 1-2 の文が成立するかを確かめ, その結果 discharge に成功している．

ここまでで，システムはゴール 1 およびその子ゴールに対して順次戦略を適用し終わっている． その後，残っているゴール2に対して再び ca 以降の戦略を順次適用する． その実行の様子は次のようになる． ここでも複数の証明対象を別々の小ゴールに分割してから， それぞれでケース分け処理を実施する． やはりケースわけは無く，処理の終わりで実施する ST + CT で， ゴール 2-1 が discharge されている．

```
[ca]=> :goal{2}
[ca] discharged: eq [lemma-1]: (s M#P Nat) = (s M#P Nat)
** Generated 2 goals
[ca]=>
:goal { ** 2-1 ----------------------------------------
  -- context module: P NAT
  -- discharged sentence
    eq [ST lemma-1]: s M#P Nat = s M#P Nat .
  -- induction variable
    M:P Nat
  -- constant for induction
    op M#P Nat : -> P Nat { prec: 0 }
  -- introduced axioms
    eq [SI lemma-1]: M#P Nat + 0 = M#P Nat .
    eq [SI lemma-2]: M#P Nat + s N:P Nat = s (M#P Nat + N) .
} << proved >>
[ca]=>
:goal { ** 2-2 ----------------------------------------
  -- context module: P NAT
  -- induction variable
    M:P Nat
  -- constant for induction
    op M#P Nat : -> P Nat { prec: 0 }
  -- introduced axioms
    eq [SI lemma-1]: M#P Nat + 0 = M#P Nat .
    eq [SI lemma-2]: M#P Nat + s N:P Nat = s (M#P Nat + N) .
  -- sentence to be proved
    eq [lemma-2]: s M#P Nat + s N:P Nat = s (s M#P Nat + N) .
}
```

以降、ゴール１の場合と同様に残りの戦略が順次適用されていく．実行の様子を下に示す.新たに説明すべき挙動は無い．

```
[tc]=> :goal{2-1}
[ip]=> :goal{2-1}
[rd]=> :goal{2-1}
[tc]=> :goal{2-2}
[tc] discharged:
  eq [TC lemma-2]: s (s (M#P Nat + N@P Nat))
      = s (s (M#P Nat + N@P Nat))
[tc] discharged the goal "2-2-1"
** Generated 1 goal
[tc]=>
:goal { ** 2-2-1 ----------------------------------------
  -- context module: P NAT
  -- discharged sentence
    eq [TC TC lemma-2]: s (s (M#P Nat + N@P Nat))
        = s (s (M#P Nat + N@P Nat)) .
  -- induction variable
    M:P Nat
  -- introduced constant
    op N@P Nat : -> P Nat { prec: 0 }
  -- constant for induction
    op M#P Nat : -> P Nat { prec: 0 }
  -- introduced axioms
    eq [SI lemma-1]: M#P Nat + 0 = M#P Nat .
    eq [SI lemma-2]: M#P Nat + s N:P Nat = s (M#P Nat + N) .
} << proved >>
[ip]=> :goal{2-2-1}
[rd]=> :goal{2-2-1}
(consumed 0.0280 sec, including 16 rewrites + 58 matches)
** All goals are successfully  discharged.
}
```

システムは全てのゴールが discharge され， 当初の証明対象が証明できたことを印字して， :auto コマンドの処理を終了している．

以上の証明において，戦略が適用されどのようにゴールが生成されたかは， **:show proof** コマンドによってみることが出来る． 下に実行例を示す．

```
root*
[si]  1*
[ca]  1-1*
[ca]  1-2*
[tc]  1-2-1*
[si]  2*
[ca]  2-1*
[ca]  2-2*
[tc]  2-2-1*
```

ゴール名の右側の * は，そのゴールに含まれるすべての 証明対象の文が discharge されていること，すなわち 証明されていることを示す．

3.1.2　交換則と結合則の証明

引き続いて，足し算 + の交換則と結合則を証明する． これら証明には先に証明した lemma-1 と lemma-2 が必要である． そのためこれらを公理として導入した新たなモジュール PNAT-L を宣言する．

```
mod! P NAT-L {
  inc(P NAT)
  eq [lemma-1]: N:P Nat + 0 = N .
  eq [lemma-2]: M:P Nat + s N:P Nat = s(M + N).
}
```

交換則の証明

まず最初に交換則を証明する．PNAT-L を文脈として設定し， ゴールを宣言する．

```
open P NAT-L .
:goal { eq M:P Nat + N:P Nat = N:P Nat + M:P Nat . }
```

今度は文脈の設定に select ではなく，open を用いた． 実行結果は次のようになる．

```
-- opening module P NAT-L.. done.

:goal { ** root ----------------------------------------
  -- context module: %
  -- axiom to be proved
    eq M:P Nat + N:P Nat = N + M .
}
** Initial goal (root) is generated. **
```

証明は帰納法を用いるため，帰納法で使用する変数を指定する．

```
%P NAT-L> :ind on (M:P Nat)
**> Induction will be conducted on M:P Nat
```

戦略として今回は :auto ではなく，陽に戦略を指定し :apply コマンドによって証明を試みる． まず最初に SI コマンドによって帰納法のベースケースとステップケースに相当するゴールを 作成する．

```
%P NAT-L> :apply  (SI)

[si]=> :goal{root}
** Generated 2 goals
[si]=>
:goal { ** 1 ----------------------------------------
  -- context module: %
  -- induction variable
    M:P Nat
  -- sentence to be proved
    eq 0 + N:P Nat = N + 0 .
}
[si]=>
:goal { ** 2 ----------------------------------------
  -- context module: %
  -- induction variable
    M:P Nat
  -- constant for induction
    op M#P Nat : -> P Nat { prec: 0 }
  -- introduced axiom
    eq [SI]: M#P Nat + N:P Nat = N + M#P Nat .
  -- sentence to be proved
    eq s M#P Nat + N:P Nat = N + s M#P Nat .
}
(consumed 0.0000 sec, including 0 rewrites + 0 matches)
>> Next target goal is ”1”.
>> R emaining 2 goals.
```

上のように2つのゴールが生成された． 現在の証明木は次のようになっている．

```
%P NAT-L> show proof
root
>[si] 1
[si]  2
```

デフォルトで次の戦略の適用対象となるゴール, すなわちターゲット・ゴールには， 証明木のノードに > と表示してそれと分かるようになっている． 次に適用する戦略として TC を指定した例を下に示す． 生成されるゴールが一々印字されるのを抑制するため, **:verbose off** としている．

```
%P NAT-L> :verbose off

%P NAT-L> :apply  (tc)

[tc]=> :goal{root}
** Generated 1 goal
(consumed 0.0200 sec, including 0 rewrites + 0 matches)
>> Next target goal is "1".
>> R emaining 1 goal.

%P NAT-L> :show proof
root
>[tc] 1
```

TCの適用によって１つの子ゴールが生成され， それが次のターゲット・ゴールとなっている． これに対して戦略RDを適用して証明を試みる．

```
%P NAT-L> :apply (rd)

[rd]=> :goal{1-1}
[rd] discharged:
  eq [TC]: 0 + N@P Nat = N@P Nat + 0
[rd] discharged goal "1-1".
(consumed 0.0160 sec, including 3 rewrites + 6 matches)
>> Next target goal is "2".
>> R emaining 1 goal.

%P NAT-L> :show proof
root
[si]  1*
[tc]  1-1*
>[si] 2
```

dischage する事が出来た．証明木のノードに * が付加されているのは それが discharge されている事を示したものである．

残りのゴール 2 に対しても TC および RD で証明を試みる． 今度はTCとRDをまとめて :apply の引数として指定する．

```
%P NAT-L> :apply (tc rd)

[tc]=> :goal{2}
** Generated 1 goal
[rd]=> :goal{2-1}
[rd] discharged:
  eq [TC]: s M#P Nat + N@P Nat = N@P Nat + s M#P Nat
[rd] discharged goal "2-1".
(consumed 0.0200 sec, including 4 rewrites + 31 matches)
** All goals are successfully  discharged.

%P NAT-L> :show proof
root*
[si]  1*
[tc]  1-1*
[si]  2*
[tc]  2-1*
```

これで，交換則の証明が完了した．

以上の証明は **:apply (SI TC R D)** のようにして続けて自動で適用するようにできる． この場合は，上の実施例のように TC を2度指定する必要はなくなる．

下の実行例で，:apply コマンドでノード指定(この場合 root)を指定している事に注意されたい．ノードを指定すると，そのノードに対して引数の戦略を適用する．この場合 root を指定しているため，これまで行った戦略適用をキャンセルし 再び証明を開始するのに等しい．

```
%P NAT-L> :apply  to root (SI TC R D )

[si]=> :goal{root}
** Generated 2 goals
[tc]=> :goal{1}
** Generated 1 goal
[rd]=> :goal{1-1}
[rd] discharged:
  eq [TC]: 0 + N@P Nat = N@P Nat + 0
[rd] discharged goal "1-1".
[tc]=> :goal{2}
** Generated 1 goal
[rd]=> :goal{2-1}
[rd] discharged:
  eq [TC]: s M#P Nat + N@P Nat = N@P Nat + s M#P Nat
[rd] discharged goal "2-1".
(consumed 0.0320 sec, including 7 rewrites + 37 matches)
** All goals are successfully  discharged.
```

上と同じことは，一旦 :select root として root をターゲット・ゴールとし， 次いで :apply (si tc rd) としてもできる.

結合則の証明

これまでと同じく，結合則も帰納法を用いて証明する． 特に新たに説明が必要なものは使われていないため， **:verbose off** として :auto で実行した例を下に示す.

```
%P NAT-L> :goal {eq (M:P Nat + N:P Nat) + P :P Nat  = N:P Nat + (M:P Nat + P :P Nat) . }
:goal { ** root ---------------------------------------
  -- context module: %
  -- axiom to be proved
    eq (M:P Nat + N:P Nat) + P :P Nat = N + (M + P ) .
}
** Initial goal (root) is generated. **
%P NAT-L> :ind on (M:P Nat)
**> Induction will be conducted on M:P Nat
[si]=> :goal{root}
** Generated 2 goals
[ca]=> :goal{1}
[tc]=> :goal{1}
[tc] discharged:
  eq [TC]: N@P Nat + P @P Nat = N@P Nat + P @P Nat
[tc] discharged the goal "1-1"
** Generated 1 goal
[ip]=> :goal{1-1}
[rd]=> :goal{1-1}
[ca]=> :goal{2}
[tc]=> :goal{2}
[tc] discharged:
  eq [TC]: s (N@P Nat + (M#P Nat + P @P Nat))
      = s (N@P Nat + (M#P Nat + P @P Nat))
[tc] discharged the goal "2-1"
** Generated 1 goal
[ip]=> :goal{2-1}
[rd]=> :goal{2-1}
(consumed 0.0320 sec, including 9 rewrites + 128 matches)
** All goals are successfully  discharged.

%P NAT-L> :show proof
root*
[si]  1*
[tc]  1-1*
[si]  2*
[tc]  2-1*
```

3.2　場合分けによる証明

先の証明の例は場合分けが必要となるものではなかった． ここでは戦略 CA を用いた場合分けによる証明を示す．

表 3.1: 可能な場合分けの組み合わせ

	$F(X:Nat)$	$G(X:Nat)$
(1)	$X \leq 7$	$X \leq 4$
(2)	$X \leq 7$	$5 \leq X$
(3)	$8 \leq X$	$X \leq 4$
(4)	$8 \leq X$	$5 \leq X$

3.2.1 モジュール FG-FUN と証明対象

下にモジュール FG-FUN の定義を示す． この例題は Maude の CITP システムの例題を CafeOBJ 用に書きなおしたものである．

```
mod! FG-FUN {
  pr(NAT)
  op F : Nat -> Nat
  op G : Nat -> Nat
  ceq[CA-1]: F(X:Nat) = 5 if X <= 7 .
  ceq[CA-2]: F(X:Nat) = 1 if 8 <= X .
  ceq[CA-3]: G(Y:Nat) = 2 if Y <= 4 .
  ceq[CA-4]: G(Y:Nat) = 7 if 5 <= Y .
}
```

特に意味のないモジュール定義である． 4つの条件付き等式が宣言されているが，それらは **CA** で始まる ラベルを持っている．これはシステムに対して，これらの等式は 場合分けのケースを網羅したものであり，これを用いて場合分けをするように 指示するものである．

証明対象とする文は次の通りである．

```
9 <= G(F(X:Nat)) + G(X:Nat) = true
```

いかなる時もこれが成立することを示すのが目標である． 上の証明対象文に含まれている $F(X:Nat)$ と $G(X:Nat)$について， それぞれが公理で宣言された 2 つのケースを持つため， 可能なケースの組み合わせは表 3.2.1のようになるはずである．

これらの組み合わせのうち，(3) のケースは X が 8 以上かつ 4 以下という条件のためあり得ない．従ってシステムはこの組み合わせについてはこれを検知し， 当該の条件を含むゴールを discharge できなければならない．

3.2.2 CITP for CafeOBJ による証明

先に示したモジュール FG-FUN を文脈として証明を実施した例を以下に示す． 個々の戦略の適用で作成されたゴールを見たいため，**:verbose on** として実施した．

また,各戦略で自動的に

```
FG-FUN> :goal { eq 9 <= G(F(X:Nat)) + G(X:Nat) = true . }

:goal { ** root ----------------------------------------
  -- context module: FG-FUN
  -- sentence to be proved
    eq 9 <= (G(F(X:Nat)) + G(X)) = true .
}
** Initial goal (root) is generated. **
```

上記のゴールに対して場合分けによる証明を行う． 使用する戦略は (CA TC RD) である． 以下システムの出力を幾つかに分割し，必要な説明を間に付加する形で示す．

```
FG-FUN> :apply  (TC CA R D )

[tc]=> :goal{root}
** Generated 1 goal
[tc]=>
:goal { ** 1 ----------------------------------------
  -- context module: FG-FUN
  -- introduced constant
    op X@Nat : -> Nat { prec: 0 }
  -- sentence to be proved
    eq [TC]: 9 <= G(F(X@Nat)) + G(X@Nat)
        = true .
}
```

最初にゴール root に TC が適用され，証明対象とする文

```
eq 9 <= (G(F(X:Nat)) + G(X)) = true .
```

の **X** を **X@Nat** に置き換えた文とした新たなゴール 1 を生成している．

次にこのゴール1に対して戦略 CA が適用され，先に見た4つのケース毎に ゴールが生成されている． CAはケースを新たな公理として導入後，それらの間に矛盾が無いかどうかを チェックする．現在は整数の順序関係の定義に矛盾が無いかどうかのみが検査される．これによってゴール1-3がdischarge されている．

```
[ca]=> :goal{1}
[le] discharged the goal "1-3"
** Generated 4 goals
```

以下ではシステムが生成した個々のゴールの内容をみる．

```
[ca]=>
:goal { ** 1-1 ----------------------------------------
  -- context module: FG-FUN
  -- introduced constant
    op X@Nat : -> Nat { prec: 0 }
  -- introduced axioms
    eq [CA]: 5 <= X@Nat = true .
    eq [CA]: X@Nat <= 7 = true .
  -- sentence to be proved
    eq [TC]: 9 <= G(F(X@Nat)) + G(X@Nat)
        = true .
}
[ca]=>
```

ゴール1-1は先の表 3.2.1に示した(2)のケースに対応する． 文脈モジュール FG-FUN で宣言されていた公理

```
ceq[CA-1]: F(X:Nat) = 5 if X <= 7 .
ceq[CA-2]: F(X:Nat) = 1 if 8 <= X .
ceq[CA-3]: G(Y:Nat) = 2 if Y <= 4 .
ceq[CA-4]: G(Y:Nat) = 7 if 5 <= Y .
```

のうち，CA-1 とCA-4 からこれらのケースが得られている．

```
[ca]=>
:goal { ** 1-2 ----------------------------------------
  -- context module: FG-FUN
  -- introduced constant
    op X@Nat : -> Nat { prec: 0 }
  -- introduced axioms
    eq [CA]: 5 <= X@Nat = true .
    eq [CA]: 8 <= X@Nat = true .
  -- sentence to be proved
    eq [TC]: 9 <= G(F(X@Nat)) + G(X@Nat)
        = true .
}
}
```

ゴール 1-2 は表 3.2.1のケース(4)に対応している．

```
:goal { ** 1-3 ----------------------------------------
  -- context module: FG-FUN
  -- discharged sentence
    eq [LE TC]: 9 <= G(F(X@Nat)) + G(X@Nat)
        = true .
  -- introduced constant
    op X@Nat : -> Nat { prec: 0 }
  -- introduced axioms
    eq [CA]: X@Nat <= 4 = true .
    eq [CA]: 8 <= X@Nat = true .
} << proved >>
```

ゴール1-3は表 3.2.1のケース(3)に対応する． 先に述べたとおり，導入された公理は互いに矛盾するため， システムはこのゴールを discharge している． discharge された証明対象であった文は， ゴールの表示上ラベル LE を追加した文として表示されている．

```
[ca]=>
:goal { ** 1-4 ----------------------------------------
  -- context module: FG-FUN
  -- introduced constant
    op X@Nat : -> Nat { prec: 0 }
  -- introduced axioms
    eq [CA]: X@Nat <= 4 = true .
    eq [CA]: X@Nat <= 7 = true .
  -- sentence to be proved
    eq [TC]: 9 <= G(F(X@Nat)) + G(X@Nat)
        = true .
}
```

CAが生成した最後にのゴール1-4は表 3.2.1に示したケース (1) に対応する．

以上のCAが生成した4つのゴールのうち，1個(1-3)は既に CA の 内部処理によって discharge されている．残りの3ゴールに対して RD が適用される．RD が行うことは文の充足性検査と矛盾の検査であった．

```
[rd]=> :goal{1-1}
[rd] discharged:
  eq [TC]: 9 <= G(F(X@Nat)) + G(X@Nat) = true
[rd] discharged goal "1-1".
[rd]=> :goal{1-2}
[rd] discharged:
  eq [TC]: 9 <= G(F(X@Nat)) + G(X@Nat) = true
[rd] discharged goal "1-2".
[rd]=> :goal{1-3}
[rd]=> :goal{1-4}
[rd] discharged:
  eq [TC]: 9 <= G(F(X@Nat)) + G(X@Nat) = true
[rd] discharged goal "1-4".
(consumed 0.0320 sec, including 35 rewrites + 117 matches)
** All goals are successfully  discharged.
```

上の実行例の通り，全てのゴールが満足されている．

以上の証明の結果に対応する証明木を表示すると次のようになる．

```
FG-FUN> :show proof
root*
[tc]  1*
[ca]  1-1*
[ca]  1-2*
[ca]  1-3*
[ca]  1-4*
```

RD はゴールを生成しないため，証明木上で新たなノードは表示されていない． 結果を確認するため，ゴール1-1を表示してみると次のようになる．

```
FG-FUN> :show goal 1-1

[ca]=>
:goal { ** 1-1 ----------------------------------------
  -- context module: FG-FUN
  -- discharged sentence
    eq [R D  TC]: 9 <= G(F(X@Nat)) + G(X@Nat)
        = true .
  -- introduced constant
    op X@Nat : -> Nat { prec: 0 }
  -- introduced axioms
    eq [CA]: 5 <= X@Nat = true .
    eq [CA]: X@Nat <= 7 = true .
} << proved >>
```

RD によって discharge された文は，ラベルに RD が追加されて表示される． これによって，RD の適用による discharge であったことを知ることができる．

参考文献

[1] Kokichi Futatsugi, Daniel Gin, and Kazuhiro Ogata. Principles of proof scores in cafeobj. Theor. Comput. Sci., 464:90–112, December 2012.