PigNose 使用手引

ver 1.0

–Draft–

澤田 寿実

info@cafeobj.org

はじめに

本書は, CafeOBJ 言語で書かれた仕様のための自動定理証明システム PigNose の利用手引である. システムは SRA版 CafeOBJ インタプリタ[1] を拡張したものになっている. 従って, これを用いるには CafeOBJ インタプリタの使用法 についても知っておく必要がある. 本書では, PigNose 特有のコマンドについて のみ説明するので, 必要に応じて CafeOBJ インタプリタのマニュアル[2]. を参照されたい. また, ここでは CafeOBJ 言語についての説明も行わない. 本書では, CafeOBJ 言語については既知のものと仮定している. CafeOBJ 言語についての解説は, 文 献[5]を参照して頂きたい.

本文は4部構成でPigNoseシステムの使用法が説明されている. 第I部でシステムのインストール法を述べ, 第II部で反駁エンジン, 第III部で詳細化検証, 第IV部で安全性モデル検査について説明する.

バグレポート/提案

システムの不具合に関する報告や, 改善提案等に関する 連絡は, 電子メイルで **info@cafeobj.org** まで頂きたい.

[1] 今の所, CafeOBJ 言語のイ ンタプリタはこれしか存在しない. システムの入手先については第 2.2.1節を参照されたい.
[2] CafeOBJ インタプリタのマニュアルは `http://cafeobj.org/download/` からダウンロード 出来る

目次

1

PigNoseシステムの概要

PigNoseの目的は，CafeOBJ言語で記述された仕様に対して resolutionベースの検証エンジンとそれを用いたさまざまな 検証用コマンドを提供する事である． 特に次の 2 点についてはそれぞれに特化された機能が 提供されている：

(1) ある仕様Mが別の仕様M'と整合的であること(M' の全て の公理が仕様Mでも成立すること)を判定する.
(2) ある仕様によって表されたシステムがある事前条件を満足してい るならば, 決して不整合な状態になったり予期しない振舞をすることがない, という安全性(safety)の検査を行うこと.

本書では(1)を**詳細化検証**, (2)を**安全性モデル検査** と呼ぶ.

これらを実現するため, PigNoseは次のような拡張をCafeOBJ処理系に対し て施した：

(a) 多ソートの一階述語論理文による公理の記述を可能とし,
(b) その上での自動定理証明機構を提供する.

(a)項はCafeOBJ言語に対する拡張と見る事も出来るがPigNoseの本来的な意図 はそこには無い．利用者の便を図ってより表現能力の高い枠組みを提供する ものである.

(b)項の自動定理証明機構は, 等号を含む多ソート一階述語論理系を対象とし, resolution 原理[1]をベースとした反駁エンジンとして実現さ れている. 反駁エンジンは独立した定理証明システムとして利用する事が可能である. これを用いる事によって, 利用者はCafeOBJで記述された仕様に対し てさまざまな検査を簡便に実行することが出来る. 反駁エンジン自体は, 既に良く知られた自動定理証明器 Otter [3]を参考にしてその機能が設計された. エンジンの中核部分の機能は基本的にOtter のサブセットと なっており，従来のCafeOBJの枠組みで記述された仕様に対して何の変更も加 えることなしに定理証明が行えるように実現されている.

以下では最初に反駁エンジンについて説明し， ついで詳細化検証およびモデル検査といった目的に特化された 各システムの使用方法を説明する.

2

インストール/起動方法

2.1 システム要件

2.1.1 プラットフォーム

2010年現在は次章で述べるようにバイナリ配布が主体となっており， 特別な事情が無い限り利用者がCommon Lisp処理系を入手してインストールす る必要はなくなっている． 以下のプラットフォームのものが提供されている:

1. i386 Linux
2. i386 MacOSX
3. win32

上記以外のプラットフォームでの利用が必要な場合はソースコードを 入手してインストールする必要がある． PigNoseは基本的にCommon Lisp処理系が稼働する環境であれば動作する．これは, SRA版CafeOBJインタプリタがCommon Lisp言語で記述されているこ とによるが, CafeOBJインタプリタはプラットフォームOSとのインタフェース 部分に処理系依存のコードが含まれているため, どのようなCommon Lisp処 理系でも動作するとは言えないのが現状である．過去2005年の時点では, 以下の処理系での動作が確認されていた(表 2.1に掲 げた稼働プラットフォームは完全なリストではない．詳細については各々の処理 系のドキュメントを参照されたい). ただし現在はAllegro Common Lisp(ver. 8.0 以降)上での開発に特化されて おり，それ以外の処理系で問題なくインストールできるか否かは検証できて いない． 上記およびそれ例外の Common Lisp 処理系についての情報は, **The Association of Lisp Users** のホームページ`http://www.lisp.org` からのリンクをたどる事で, 容易に入手できる．

Common Lisp 処理系	稼働プラットフォーム
GCL(version 2.3以上)	i386 Linux, BSD Sun Sparc, Sun OS 5(gcc)
Allegro Common Lisp (ver5.0以上)	Linux Windows95/98/2k?
CMU CommonLisp	Sparc Slaris i386 Linux
CLISP	i386 Linux

表 2.1: CafeOBJ インタプリタの稼働環境

2.2 インストール方法

2.2.1 配布形式

CafeOBJインタプリタは過去にシステムはソースとバイナリ両形式で配布され ていたが，現在はバイナリ形式による配布が中心である[1] 先に述べたとおり，バイナリ形式はi386Linux，i386 MacOSX, と Windows(win32)の3つのプラットフォームのみである．現在配布されているCafeOBJインタプリタ(ver.1.4.6以降)はPigNose拡張を含 んでいるためそれを入手すれば良い． インタプリタはCafeOBJの公式ホームページ

http://www.ldl.jaist.ac.jp/cafeobj/

からダウンロード出来る．

ソース形式は Unix のテープアーカイブ形式(TAR)ファイルを gzip によって固 めたものである．これには CafeOBJ インタプリタ自体のソースも含まれてい る．上記のサイトから同様にダウンロード可能である．

2.2.2 ソースからのインストール方法

以降ではソース配布のファイルを単に**ディストリビューショ ン**と呼ぶ．これを展開するとインストールに必要なソースファイルやライ ブラリ等を格納したディレクトリが生成される． 例えば Unix 上では次のようにして展開する：

```
% gunzip -c cafeobj-1.4.6.tar.gz | tar xvf -
```

この例の場合は**cafeobj-1.4.6**という名前のディレクトリが 生成されるはずである． 以下展開して作成されたディレクトリの下でインストール作業を 実行する事となる．

Unix/Linux 上でのインストール方法

(A) Allegro Common Lispの場合 使用するCommon Lisp処理系が Allegro Common Lisp(ACL — ver.8 以降)の場合にはソースコードからのイ ンストールが非常に楽である．次のように行う．

[1] ソースコードも配布されているが，バージョンが古いもののみである． 今後はバイナリに同期して配布する予定である．

1. ソースを展開したディレクトリへ移動する.

```
% cd cafeobj-1.4.6
```

2. Allegro Common Lisp を起動する.

```
% alisp
```

3. ACLへ以下を入力する(**CL-USER(1):**はACLのプロンプトである).

```
CL-USER(1): :ld make-deliv.cl
```

これによってソースコードからのシステムの構築が開始される. これが終了したら**dist**というディレクトリの下にCafeOBJ インタプリタが構築されているはずである.

4. **dist** へ移動する.

```
% cd dist
```

そこにはcafeobj-1.4という名前のディレクトリがある. ここ以下にインタプリタ本体及び実行時に必要となるライブラリなどが 格納されている.

5. 必要に応じて **cafeobj-1.4** を適当なディレクトリに移動もし くはコピーする.

```
% tar cf - ./cafeobj-1.4 | tar xvf - -C /usr/local
```

上の例ではインタプリタを**/usr/local**の下にtarコマンドを 使用してコピーした(いろいろな事情によりコピーを行う場合は **cp** コマンドは避けたほうが良い).

コピーや移動先に書き込む権限がない場合は，**sudo** コマンドを 用いるか，それを使う権限も無い場合は，権限を持っているユーザに依頼する. インストール先は任意で良いので特に移動する必要は無い.

6. インタプリタの実行コマンドは**cafeobj-1.4/bin/cafeobj**である. これを**PATH**が通っているディレクトリにコピーもしくは移動, あるいはこのファイルにシンボリックリンクを貼る.

```
% cd /usr/local/bin
% ln -s /usr/local/cafeobj-1.4/bin/cafeobj .
```

上の例では**/usr/local/bin**に，実行コマンへのシンボリック リンクを張った.

7. **cafeobj-1.4**を**/usr/local/**へインストールした場合はこ れで終了である. それ以外の場所へインストールした場合はインタプリタ本体の場所を 設定する必要がある. これには次の2つの方法がある:

   a) **cafeobj**コマンドを修正する方法. **cafeobj**コマンドはshellスクリプトである. この中身は次のようになっている.

   ```
   exec ${CAFEROOT:-"/usr/local/cafeobj-1.4"}/lisp/CafeOBJ -- $*
   ```

   この **"/usr/local/cafeobj-1.4"**の部分をインストール先の ディレクトリ名に修正すればよい. 例えばインストール先ディレクトリが **"/opt"**であれば，上記を

   ```
   exec ${CAFEROOT:-"/opt/cafeobj-1.4"}/lisp/CafeOBJ -- $*
   ```

   のように修正すればよい.

   b) 上の**cafeobj**コマンドの中身を見てわかる通り, 環境変数**CAFEROOT**が設定されていればそれをインストール先 ディレクトリ名として採用する. 例えば使用しているshellが bashの場合は

   ```
   export CAFEROOT=/opt/cafeobj-1.4
   ```

   のような記述を**.bashrc**などに記載しておけばよい.

(B) その他のCommon Lisp処理系の場合　Allegro Common Lisp以外の処理系でソースコードからインストール する場合は以下の手続きに従う[2]． Unix (Linux) 上でのインストールは以下のように行う．

1. ディストリビューションを展開したディレクトリへ移動する．
```
% cd cafeobj-1.4.6
```
2. 使用する Common Lisp 処理系や, インストール先のディレクトリの 設定を行う．これにはディストリビューションに含まれている **configure** コマンドを次のようにして起動することによって行う:
```
% ./configure [--with-lisp=<Lisp処理系指定>] \
  [--prefix=<インストール先>]
```
ここで, <Lisp処理系指定> はベースとして使用する Common Lisp 処理系 の指定であり, 以下のものの中から指定する：

(1) **gcl** — GCL
(2) **acl** — Allegro Common Lisp (ver 5.01 以下)
(3) **acl6** — Allegro Common Lisp (ver 6.x)
(4) **acl7** — Allegro Common Lisp (ver 7.x)
(5) **acl8** — Allegro Common Lisp (ver 8.x)
(4) **cmu-sparc** – CMU Common Lisp, Sparc Sun OS
(5) **cmu-pc** — CMU Common Lisp, i386 マシン
(6) **clisp** — CLISP

特に指定を行わなければ gcl がデフォルトで選択される．

<インストール先> は, システムをインストールするディレクトリの パス名を指定するものである．特に指定が無ければ, デフォルトで **/usr/local** がインストール先となる．例えば, 下に示す例では Lisp 処理系として Allegro CL(ver8.x) を 指定し, インストール先は既定値(**/usr/local**)として構成している．
```
% ./configure --with-lisp=acl8
```
3. make コマンドによって, システムの構築/インストールを行う．
```
% make bigpink
% make install-bigpink
```
最初の make コマンドの発効によって, システムのコンパイルが 行われ, 次の make コマンドの発効によって, システムが指定の ディレクトリにインストールされる．

Windows 上でのインストール方法

使用するCommon Lisp処理系がAllegro Common Lisp(ver.8.x)の場合は 上で記載したUnix上でのインストール方法と同様である．

それ以外のCommon Lisp処理系についてはソースからインストールを 行う上で一般的な枠組みが用意されていないのが現状である．

[2]現在ここで記載した通りにできるかどうかは検証されていない． もしこのとおり実行してうまくいかない場合はご一報いただけると 幸いである．ついでに解決方法も知らせていただければなおありがたい．

2.2.3　バイナリ配付のインストール方法

バイナリ配付の場合, インストールされるインタプリタはスタンドアローン, つまり実行時に Common Lisp 処理系を必要としない. TODO

i386 Linux

Windows

2.2.4　何がインストールされるか

インストール先が CAFEROOT であったとすると, 次の ようになる：

- CAFEROOT/bin/cafeobj：検証推論システムの組み込まれた CafeOBJインタプリタの起動コマンド.
- CAFEROOT/cafeobj-1.4：インタプリタのライブラリ等を格納するディ レクトリ. このディレクトリの下は以下のようになる
  - exs：CafeOBJ の例題モジュールファイルが格納される
  - lib：検証推論システムのために必要なモジュールファイル “fopl.mod” が格納される.
  - bin：インタプリタの本体が格納される.
  - prelude：CafeOBJ インタプリタの初期化ファイルの格納場所. 以下のファイルが置かれる.
    * std.bin：CafeOBJ インタプリタが動作する上で必要となる初期設 定ファイル.
    * site-init.mod：利用者用の初期化ファイル.

    これらはインタプリタが起動する毎に最初に読まれる.

2.3　起動方法

PigNose が組み込まれた CafeOBJ インタプリタの起動方法は, 通常の CafeOBJ インタプリタと全く同じである. 以下では Unix/Linux の場合, および Windowsのそれぞれの場合について簡単に説明する. インタプリタとの対話方法の詳細, および CafeOBJ 自体については CafeOBJ イ ンタプリタのマニュアル [4]を参照されたい. また CafeOBJ 言語に ついては [5]を参照されたい.

Unix/Linux 上の場合　コマンド “cafeobj” によってインタプリタが起動される. Emacs あるいは, XEmacs を使っている場合は, ディストリビューションに付属 しているcafeobj-mode パッケージを用いてインタプリタと対話できる. 詳細は , ディストリビューションの elisp/cafeobj-mode.el を参照されたい.

Windows 上の場合　ソース配付からインストールした場合は, インストール手続きによって 作成された cafeobj.dxl をダブルクリックすることで, インタプリタが起動さ れる.

バイナリ配付の場合は, 独立したアプリケーションとして CafeOBJ.exe が インストールされるので, これを起動すればよい.

2.3.1 fopl.mod のロード

検証推論システムが動作するためには, “fopl.mod” というファイルをインタ プリタにロードする必要がある. このファイルにはPigNoseの機能を使用する ために必要となる新規の組み込みモジュールFOPL-CLAUSEの定義や, その他の 環境設定のためのCafeOBJスクリプトが記述されたファイルである. CafeOBJには自動ロード機構があり, 組み込みモジュールFOPL-CLAUSEが初め て参照された時に自動的に**fopl.mod**がロードされるように予め設定 されている. このため通常は手で**fopl.mod**をロードする必要はない.

あらかじめロードしておきたい場合は, インタプリタを起動した後に,

```
CafeOBJ> require fopl
```

のようにして, require コマンドによってロードできる. いちいちこれを入力する手間を省くため, システム初期化ファイル (site-init.mod)に上のコマンドをいれて置くと便利である. Unix(Linux)上で利用する場合は, ホームディレクトリの直下に **.cafeobj** と いう名前のファイルがあると，CafeOBJインタプリタは起動時にこれを初期 化ファイルとして書かれている内容を実行する. 上記の site-init.mod を 用いずにこのファイルに “require fopl” をいれて置いても良い.

3

反駁エンジン

本章ではPigNoseシステムの核である反駁エンジンについて説明する． 本章の内容は読者が導出原理(resolution principle)に基づいた定理自動証明 についての基本的な知識を有する事を仮定している． 付録 A に簡単な解説を載せたが， 不案内な読者は文献([1]等)を参照して欲しい． また付録 C に ここで使い方を説明する反駁エンジンが使用している各種の推論ルールについ て概説があるので必要に応じてそれも参照されたい．

3.1 反駁エンジンの機能

最初に述べた通り反駁エンジンは等号を含む一階述語論理系の上での定理証明 機能を提供するものであるが, これをもう少し具体的に述べると次のようになる：

- CafeOBJ モジュールで宣言された公理を，等号を含む一階述語論理(以降 これを **FOPL**と略称する)の論理式とみなし，
- そのモジュールの文脈において，与えられた述語がモジュールの公理か らの論理的帰結となるかどうかを反駁法によって証明する．

前述のように反駁には導出原理 (resolutio principal) [1]を 用いるが, 等号を扱うための paramodulation 機構(C.3章)が 推論ルールとして含まれている．導出に用いられる推論ルールには基本的な binary resolution (C.1章) に加えて, (negativ/positive) hyper resolution (C.2章), また unit resulting resolution が用意されてい る．また, 論理式に含まれる関数適用フォームの簡単化機構として demodulation(C.4章)が組み込まれている．また推論戦略としては Set of Support 戦略(SOS 戦略) を用いている[1]．

反駁エンジンを組み込んだインタプリタは, 通常の CafeOBJ インタプリタとし てなんら変わる事無く使用できる．エンジンの機能は新規の言語要素，コマンド 群として提供される．

[1] SOS戦略及び反駁エンジンの動作については 第 3.9 に概要を示した．

3.2 CafeOBJモジュールの解釈

3.2.1 FOPL文

PigNoseにおけるFOPL文は通常の項と同様に CafeOBJ 項である．これは組み込みモジュール FOPL-CLAUSE(第 3.3章 を参照)で定義されている(FOPL文の構文の詳細は第 3.4章 で説明する)が, これは従来のCafeOBJ言語による記述と出来るだけ境界なし にFOPL文を使えるようにするための方策である．これを具体的に実現するため,

CafeOBJ組み込みのソート Bool を, FOPLの真偽値を表現するためのソートとし て使用する．

という方針がとられている．ソートBoolは論理演算を定義している組み込み モジュールBOOL[2]で定義されているモジュールであり，CafeOBJでは 特に指定の無い限りBOOLモジュールは利用者が定義した全てのモジュールに 暗黙的にインポートされている．

このことから言えるのは

ソートBoolのCafeOBJ項はFOPLの文(述語)と解釈される.

という事である． その他のCafeOBJ項(組み込みソートBool以外の項)は述語の引数として出現す るのでない限り, FOPLでの正当な式とは見なされない．またCafeOBJ モジュールで記述された公理は, この事を基本として以下で述べ るように解釈される．

3.2.2 等式の解釈

CafeOBJ の(条件付き)等式は, 次のような形をしている[3]

```
        <左辺> = <右辺> if <条件> .
```

これをFOPLのimplication(含意 ->)

```
        <条件> -> <左辺> = <右辺>
```

と解釈する．またこれは下と等価である．ここで, ~はFOPL文の否定を表す論理演算である． また | は論理和を表すFOL文表現である (表 3.1 を参照)．

```
        ~<条件> | <左辺> = <右辺>
```

[2]実際にはBOOLからインポートされるTRUTH-VALUEで 定義されている．

[3]CafeOBJ には **eq** または **ceq** で宣言される 可視等式と，**beq** または **cbeq** で宣言される 振舞等式の2種があるが，ここではそれらを区別せずに扱う．

ここで, CafeOBJ等式の=をFOPLの等号=と解釈している 事に注意. resolution を行う際には, 最後に得られた形の式を節形式[4] に変換して用いる. これはシステムが自動的に行うので, 利用者はこれに関して特に気 を使う必要はない. 等式の＜条件部＞が省略されている場合は, 単純に

```
＜左辺＞ = ＜右辺＞
```

というFOPLの等式となる.

3.2.3 組み込み述語の解釈(変換)

先にCafeOBJのソートBoolの項は, 自動的に FOPL 文とみなされると述べた. CafeOBJにはBoolを値のソートとするオペレータがいくつか組み込まれている. これらは組み込みモジュールBOOLとそのサブモジュールで宣言されて いるものであるが, 表3.1に示したようにそれらと 意味的に対応するFOPL論理結合子へとシステムが変換する.

CafeOBJ組み込み述語	FOPL論理結合子
and	_&_ (論理積)
_or_	_|_ (論理和)
not_	~_ (否定)
implies	_->_(含意)
iff	_<->_ (同等)
and-also	_&_ (論理積)
or-else	_|_ (論理和)
_==_	_=_ (等号)
_=b=_	_=_ (等号)

表 3.1: 組み込み述語の解釈

また組み込みの _xor_(exclusive or) に関しては,

```
p xor q --> (~p | ~q)&(p | q)
```

のような等価変換を行う. なお具体的なFOPLの文法については第3.4章で説明 する.

3.2.4 制約事項

CafeOBJでは公理として条件付き等式ばかりではなく遷移規則も記述 可能であるが，これを適切に扱うことは不可能である[5] このため PigNose では遷移規則による公理は無視される. これと併せて, PigNose を利用するに当たっては以下に掲げるような制約事項が ある：

[4]述語論理における節形式とは 0 個以上のリテラルの集合であり， リテラルの論理和の全称閉包の事である．また，リテラルとは 素式(原子論理式)またはその否定を言う．例えば，節形式 $\{P(x), \neg Q(x, y)\}$ は, $\forall x \forall y (P(x) \vee \neg Q(x, y)$ を意味する．

[5]遷移規則 $t \Rightarrow t'$ を方向つけられた関係とし， 遷移関係の性質を公理によって表現することにより ある程度の表現は可能であるが，システムにとって 遷移規則の持つ推移則を適切に扱うことは困難である．

1. 遷移規則(**trans, ctrans** によって宣言された公理)は使用できない.
2. 利用者定義のオペレータが組み込みソートBoolを引数に持っていてはな らない.
3. 振舞等式の=と可視等式の=とを区別しない.
4. **if_then_else_fi** は使用できない.
5. メンバシップ述語 (**_:<ソート名>**) は使用できない.
6. 組み込みの等式(Lisp関数で定義されているもの)はresolutionに 用いることはできない. システムはこれらを無視する. 但しdemodulator(第 C.4章)としては使用できるので, システ ムは自動的にこれらをdemodulatorとして使用する.

3.2.5 その他の CafeOBJ 組み込みモジュールの扱い

BOOLモジュールの扱いについては第 3.2.3 節で述べたが, ここでは, NATやSTRING等のその他の組み込みモジュールの一般的な扱いにつ いて述べる. 現在 CafeOBJ インタプリタには以下の組み込みモジュールが提供 されている：

```
NZNAT       NAT       INT       RAT       FLOAT     STRING
CHARACTER   2TUPLE    3TUPLE    4TUPLE    PROPC
```

これらのモジュールのうち, PROPCを例外として他の全ての組み込みモジュール に含まれる公理は次のような取り扱いを受ける：

- 全ての公理は(組み込みの)demodulatorとしてのみ使われる.
- 従って推論ルールを用いた新たな節の導出には用いられない.
- 条件付きの等式は無視される.

モジュール**PROC**をサブモジュールに持つモジュールにおいて PigNoseはうまく機能しない.

3.3 組み込みモジュール – FOPL-CLAUSE

あるモジュールで反駁エンジンの機能を利用するには, そのモジュール 文脈で**FOPL-CLAUSE**という名前の組込みモジュールがサブモジュール の一つになっている事)が必要である. このモジュールはシステムのライブラリファイル**fopl.mod**で定義さ れ, FOPLの論理式の構文が定義されている. またシステムは, FOPL-CLAUSE が輸入されているモジュールではPigNoseの反駁エンジンで導入された機能が使用できるように実行環境を設定する[6] .

3.3.1 FOPL-CLAUSE の自動輸入

モジュール FOPL-CLAUSE を簡便に参照するために, 新規のCafeOBJスイッチ **iclude FOPL-CLAUSE** が用意されている.

[6]第 2.3.1 節で述べた通り,fopl.mod は, システム の初期化ファイル site-init.mod に, **require fopl** というコマン ドをいれておく事によって, システム(CafeOBJ インタプリタ)を立ち上げる毎 に自動的にシステムにロードされるようにする事ができる.

```
set include FOPL-CLAUSE on   -- 自動輸入を有効にする（初期値）
set include FOPL-CLAUSE off  -- 自動輸入を無効にする
```

このフラグを**on**に設定する事によって， モジュール内で**ax**あるいは**goal**による公理宣言 (第3.5章を参照)が出現した場合，自動的に FOPL-CLAUSE がそのモジュールに輸入されるようシステムが振舞う． スイッチの初期値は**on**であり，特に理由が無い限りこれを **off**にする必要は無い．

3.4 FOPL の構文

ここではモジュール**FOPL-CLAUSE**で定義されるFOPLの構文を示す． FOPL の文はCafeOBJ項で表現されることは既に述べた． そのソートは**FoplSentence**と名付けられ， 下で示したCafeOBJソート宣言文に示されているように, 組み込みソート Bool の上位ソートとして宣言されている．

FOPL文のソート宣言

```
      [Bool < FoplSentence]
```

この定義によってBoolのCafeOBJ項はFOPLの文となり， またBool値のオペレータはFOPLの述語とみなされる．

3.4.1 論理結合子の構文

論理結合子の構文をCafeOBJのオペレータ宣言で示すと以下のようになる (先に掲げた表 3.1も参照)．

論理和

```
op _|_ : FoplSentence FoplSentence -> FoplSentence {prec: 107}
```

論理積

```
op _&_ : FoplSentence FoplSentence -> FoplSentence {prec: 101}
```

否定

```
op ~_ : FoplSentence -> FoplSentence {prec: 0}
```

含意

```
op _->_ : FoplSentence FoplSentence -> FoplSentence {prec: 120}
```

同値

```
op _<->_ : FoplSentence FoplSentence -> FoplSentence {prec: 120}
```

等号

```
op _=_ : FoplSentence FoplSentence -> FoplSentence {prec: 51}
```

次に，限量子に関しては下のように定義されている．

全称限量子(forall)

```
op \A[_] _ : VarDeclList FoplSentence -> FoplSentence {prec: 125}
```

存在限量子(exists)

```
op \E[_] _ : VarDeclList FoplSentence -> FoplSentence {prec: 125}
```

ここで, **VarDeclList**は変数名あるいはon-the-flyの変数宣言 (**VAR:SORT**の形の変数宣言 – **VAR**は変数名, **SORT**はソート名である) をコンマ記号(,)で区切って並べたものである．

3.4.2 述語・命題の宣言

先に述べた通り, ソート**FoplSentence**の項がFOPL 文であり, 前節で示した論理結合子や限量子を用いて記述される． またBool値のオペレータは一般に述語として解釈されるので, 述語や命題は通常のCafeOBJ のオペレータ宣言によって行うことが できる．命題の場合は, 引数を持たない Bool の定数項として宣言すれば良い．

このためには, 下のようにCafeOBJの**pred**構文を用いるのが 便利である：

```
pred P1 : S1 S2          -- 2引数の述語
pred P0 :                -- 命題
```

3.4.3 FOPL文の記述例

下に FOPL 文による論理式の記述例を示す．これは 組み込みの NAT を文脈モジュールとして使用したものである．通常記号論理で使用される構文とほぼ同様の記法で論理式が記述 できるようになっていることが了解されるであろう[7]．

[7]ここで例示したように，あるモジュールで反駁エンジンを用いて何か作業を おこないたい場合は，**open** コマンドによってそのモジュールを 文脈にするとともにオペレーションや公理の追加が可能な状態にし， 陽に**protecting(FOPL-CLASE)**とすることによって PigNoseの 機能を利用することもできる．

```
-- NAT を文脈として用いる
open NAT
-- FOPL 文の記述を可能とするために FOPL-CLAUSE を輸入する
protecting(FOPL-CLAUSE)
-- 命題, 述語の宣言
pred p : .
pred q : .
pred r : .
pred P : Nat .
pred Q : Nat .
pred R : Nat Nat .
pred S : Nat Nat .
-- Nat の 定数
op  a   : -> Nat .
-- t1 〜 t5 は CafeOBJ の let 変数.
--
let t1 = (p -> q) -> (q -> r) .
let t2 = \A[X2:Nat]\E[Y1:Nat]\A[X1:Nat]\E[Y2:Nat]R(X1,Y1) & S(X2,Y2) .
let t3 = (\A[X:Nat]P(X) -> (\E[Y:Nat]R(X,Y))) &
         (\A[X:Nat]~ P(X) -> ~ (\E[Y:Nat]R(X,Y))) .
let t4 = (\A[X:Nat]P(X) -> (\E[Y:Nat]R(X,Y))) |
         (\A[X:Nat]~ P(X) -> ~ (\E[Y:Nat]R(X,Y))) .

let t5 = \A[X:Nat]P(X) ->
                  (\E[Y:Nat](R(X,Y) -> P(a)) &
                            (\A[Z:Nat]R(Y,Z) -> P(X))) .

-- show term t1 等とすることによって, システムがパーズした
-- 項を表示することができる.
show term t4 .
```

3.5　FOPL 文による公理の宣言

FOPL文による公理の宣言を可能とするため， 次に示す新規構文が導入された. これらは反駁エンジンの実行の際にその他の CafeOBJ の通常 の公理と併せて推論に用いられる. ただし, CafeOBJ の簡約コマンド (**reduce**, **breduce**, **exec**) で用いることは出来な い. 反駁エンジンによる定理証明にのみ有効である.

1. `ax <ラベル> <項> .`

・ **ax**は FOPL 文による公理を宣言するものである.

- **<項>**はソート FoplSentence の項でなければならない.
- **ax**で宣言された公理は従来の CafeOBJ の **red**コマンドや **exec**コマンドで使用される書き換え規則と して使う事は**できない**.
- **<ラベル>**は公理のラベルであり, 通常の CafeOBJ の公理のラベル と同じ構文とし同様に省略可能とする.
- <項> の中に自由変数が出現する場合は，それらは暗黙的に 全量限定子∀で束縛されているものと解釈する.

2. `goal <ラベル> <項> .`

goalは**ax** と同様であるが，**<項>** の否定を自動的 に取るものであり, "**ax** ~ **<項>**"の簡易表記法である.

axと**goal**は従来のCafeOBJの公理宣言文(等式や遷移規則) が出現できる場所ならばどこにでも出現して良い. また, CafeOBJ のモジュール表示コマンド(**show <モジュール式>**)によって モジュール内容が表示される際には, **ax**や**goal**による FOPL 文も表示されるようになる.

FOPL文による公理の記述例　下は FOPL 文による公理宣言の例である. ここでは整数(Int)上の大小関係(<=)についての 公理の幾つかが FOPL 文によって記述されている.

```
mod! INT* {
  [ Int ]
  op 0 : -> Int
  op _+_ : Int Int -> Int
  op _-_ : Int Int -> Int
  pred _<=_ : Int Int

  vars M N : Int
  ax M <= M .
  ax 0 <= M & 0 <= N -> 0 <= M + N .
  ax M <= N -> 0 <= N - M .
}
```

上の例では ax による公理宣言全てに自由変数 M あるいは N が出現している. これらは先に述べた通り, 全称限量子によって暗黙的に束縛される. 従って例えば

```
ax 0 <= M & 0 <= N -> 0 <= M + N .
```

は,

```
ax \A[M,N] 0 <= M & 0 <= N -> 0 <= M + N .
```

と同等である.

3.6 反駁エンジンの新規コマンド群

ここでは新規に導入されたコマンドの全てをまとめて提示する． 個々のコマンドの詳細については後の章で順次解説する．

flag コマンド 反駁エンジンの証明戦略の設定や, エンジンの動作を制御するためのさまざまな フラグを設定するためのコマンドである. 構文は次のとおり.

```
<flagコマンド> :: = flag(<フラグ名>, { on | off })
```

第一引数の**<フラグ名>** は設定したいフラグの名前であり，第二引数で フラグの on/off を設定する．フラグの種類やその意味については後述する (第 3.11章).

param コマンド flag コマンドと同様に，エンジンの動作を制御するためのパラメータの設定を 行うためのコマンドである.

```
<paramコマンド> ::= param(<パラメータ名>, <値>)
```

<パラメータ名>は値を設定したいパラメータの名前，**<値>** は設 定したい値であり，整数値を指定する．設定可能な値の範囲は個々のパラメータ 毎に定められている．パラメータの種類や意味については後述する (第 3.12 章).

save-option コマンド 現在設定されているフラグとパラメータの値を名前を付けて保存するためのコマ ンドである.

```
<save-option コマンド> ::= save-option <オプション名>
```

指定の**<オプション名>**でフラグとパラメータの現在値を保存する． **<オプション名>** は任意の英数文字列． 詳細は 3.10.1章で説明する.

option コマンド フラグやパラメータの値を全て初期値に戻したり，save-option コマ ンドで以前に保存されているフラグ/パラメータ値を現在値として設定するた めのコマンド.

```
option { reset | = <オプション名> }
```

<オプション名>は, 以前に save-option コマンドを用いて保存して おいたオプションの名前を指定する. **reset**を指定すると，全ての値が初期値に再設定される． 詳細は 3.10.1章で説明する.

db reset 反駁エンジンは, 文脈モジュールで推論を実行するにあたって，さまざまな情 報を維持し管理する．‘**db reset**’ はこのデータベースの初期設定を行 うためのコマンドである(CafeOBJ モジュールの公理を節形式に変換する のは, このコマンドが発せられたタイミングで行われる.) 通常は推論実行コマンド ‘**resolve**’ の直前 でこれを行う必要がある．自動モード(フラグ auto あるいは auto3 を on) で推論を行う場合は不要である.

list コマンド フラグやパラメータ，また推論に用いる節集合(sos や usable – 後述) の内 容を表示するためのコマンドである.

```
<listコマンド> ::= list { axiom | sos | usable | flag |
                          param | option | demod }
```

それぞれの引数の値に対して表示される内容は次の通り：

axiom	:	文脈モジュールで宣言されている公理を節形式で印字
sos	:	節集合 SOS に含まれる節
usable	:	節集合 Usable に含まれている節
flag	:	フラグの一覧と現在の設定値
param	:	パラメータの一覧と現在の設定値
option	:	save-option コマンドでセーブされたオプション名の一覧
demod	:	demodulator 一覧

注意: flag, param, option の3つの場合を除き, list コマンドが有効となるのは, 先に述べたシステム初期化 コマンド db reset が実行された後である. また, demod に関しては, db reset を実行時にシステムが生成した demodulator のみが表示される. システムはこの時点では組込みの等式(右辺側がLisp関数で記述されている 等式)のみを, demodulator として登録する. それ以外の公理については システム実行中に demodulator とされるので, この時点で知る事は出来ない.

sos コマンド 節集合 SOS を設定するためのコマンド. あらかじめ db reset が実行されて いる必要がある. 構文は下の通り：

```
<sosコマンド> ::= sos { = | + | - } <節集合>
```

第一引数の意味は以下の通り：

=	:	sos を **<節集合>**で指定された節の集合に設定する
+	:	現在の sos の内容に **<節集合>**で指定された節を加える
-	:	現在の sos から **<節集合>** で指定された節を削除する

また, **<節集合>**の構文は下の通りである：

```
<節集合> ::= '{' <節指定> { , <節指定> }* '}'
```

<節指定>は，公理ラベル, 節識別子，あるいは**let**変数名の いずれかである. 公理ラベルは, CafeOBJ の公理宣言で指定された, 公理の ラベル. 節識別子はシステムが節を生成する際に与えられるものであり，list axioms コマンド等によって知る事ができる. let 変数名が指定された場合は， 指定の変数に束縛されている項を節形式に変換したものが用いられる.

節指定の解釈は, 次の規則に従うものとする:

1. 節指定が数字の場合, これは節識別子であると解釈する.
2. 数字では無い場合, まず公理のラベルであると解釈する.
3. 2 で該当する公理が一つも無い場合は, let 変数名であると解釈する.

上のいずれの解釈によっても, 節が見付からない場合はエラーとする. 節指定として公理のラベルが指定された場合，同じラベルを持った公理が 複数存在する場合は，それらの公理全てが指定されたものと解釈する.

SOS 集合は初期には空であり，また **db reset** を行った時点で空 に設定される． **sos**コマンドの実行に伴って，usable 集合の内容が副作用として 決められる． つまり，**list axioms**で表示される節から SOS 集合の内容を取り 去ったものがusable 集合として設定される.

resolve コマンド 反駁エンジンを起動するためのコマンド． 本コマンドを発する事によって, 反駁エンジンによる定理証明プロセスが 開始される.

```
<resolveコマンド> ::= resolve {. | <ファイル名>}
```

引数が**<ファイル名>**で指定されるファイルへのパス名の場合は，指 定のファイルへ実行ログが出力．これが '**.**' (ピリオド記号)の場合は標準出力へ ログが出力される.

clause コマンド 指定された項を Skolem 標準形の節形式に変換し, 結果を印字する. 試験の ためのコマンドである.

```
<clauseコマンド> ::= clause <項> .
```

<項>はソート**FoplSentence**の項でなければならない． 指定 の項を節形式に変換して印字する.

show/describe コマンド 従来のCafeOBJコマンド**show**と**describe**を拡張したものである.

```
<showコマンド>      ::= show <節ID>
<describe コマンド> ::= describe <節ID>
```

どちらの場合も **<節ID>**で指定される節を印字する．**<節ID>**は節の識別子であり， '**list axioms**' や，'**list sos**' 等のコマンドで表示さ れる内容から知ることができる． **describe**の場合は, 指定の節がCafeOBJモジュールの公理を節形式に 変換して得られた節であった場合, 対応する公理がどれであるかの情報も 併せて印字する.

lexコマンド lrpo による項の大小比較を行う際に用いられる演算子の順序関係を定義する ためのコマンドである. このコマンドについては, 第 3.16.2 節で説明する.

3.7 PigNose の一般的なスクリプト構造

システムの詳細な使用法は引き続く各章で説明されるが, ここでは簡単な例を示して典型的な使用方法について述べる. 下は通常想定される典型的な使い方の大枠を示したものである：

(1) 検証対象とするモジュールをオープン
これは実行文脈となるモジュールを設定するのが主目的であるが, 証明を実行する際には, 通常付加的なオペレータや 公理等の追加宣言が行われると想定される. 例えば, 証明対象(ゴール)とする FOPL 文は, 追加公理として動的に宣言されることが多いと思われる. このためには, オープン文を用いるのが便利である.

(2) (必要に応じて) **FOPL-CLAUSE** をインポート
証明実行の文脈となるモジュールのサブモジュールに**FOPL-CLAUSE** が既に輸入されているのであれば, これは不要である.

(3) (必要に応じて)オペレータや公理等の追加宣言
(1) で述べた通り.

(4) 反駁エンジンの起動

(4-1) エンジンの実行環境の設定
フラグ/パラメータの設定を行ったり, マニュアルモードの場合は, システムの初期化(db reset), SOS 節集合の設定等を行う.

(4-2) 証明実行

resolve コマンドを発行し, エンジンを起動する.

下に極めて簡単な例を示す. ここでは自動モードを用いた場合と, 手動で推論ルールを 設定する場合の2つの例が示されている. いずれも単純な例であるが, 上で述べた通常の典型的と思われるスクリプトの構造を反映したものである.

```
-- FOPL-CLAUSE が必要に応じて自動的に輸入されるように設定する
set include FOPL-CLAUSE on
--
module! TEST1 {
  [ Human < Life ]
  pred mortal : Life
  op Socrates : -> Human
  -- 生あるものは死す
  ax \A[X:Life] mortal(X) .
}

**>  典型的なスクリプト構造

**> (1) 対象とするモジュールをオープンする
open TEST1

**> (2) FOPL-CLAUSE は既に輸入されているので改めて輸入する必要はない.
**> (3) 必要に応じて公理などの追加宣言
-->     ここでは証明したいことを追加公理で宣言.
-->     これは ax ~ motal(Socrates) と等価.
goal mortal(Socrates) .

**> (4) 反駁エンジンの起動
**> (4-1) PigNose 実行環境の設定
-->       オプション(フラグ/パラメータ)をリセット.
-->       対話型のシステムなので, option reset を習慣付けることが大事
option reset

-->        自動モードで実行する
**> auto mode
flag(auto, on)

-- **> このフラグをセットすると沢山の情報が出力される, try it!
-- **> flag(very-verbose,on)

-->         証明は1つで良い
param(max-proofs, 1)

**> (4-2) 証明実行
-->       エンジンを起動する
resolve .

**> 文脈モジュールをクローズ : 後始末
close
```

下はマニュアルモードの場合である. 例題は上のオートモードの場合と同じ TEST1 を想定している.

```
**> 次はマニュアルモードによる例 ****************************
**> manual mode

**> フラグ/パラメータを初期状態に戻す
option reset

**> 文脈モジュールをオープンする
open TEST1

**> 証明したい文を宣言：今度は公理にラベルを指定している
--
goal[GOAL]: mortal(Socrates) .

**> マニュアルモードなので，システムの初期化を陽に実行する
**> 必要がある
db reset

**> SOS 節集合の設定
sos = {GOAL}

**> 推論ルールとして negative hyper resolution を
**> 用いる
flag(neg-hyper-res, on)

**> エンジンの起動
resolve .

**> 文脈モジュールのクローズ
close
```

3.8　節の印字形式

システム実行中, あるいはコマンドによって さまざまな情報の表示や証明木の印字などで節が印字される. システムの動作を見たり, 結果を調べるためには節の印字形式に付いて 知っておく必要がある.

3.8.1　節表示の一般形式

一般に節は次のような形式で表示される.

```
<節番号>:[<導出履歴>] FOPL文
```

ここで, **<節番号>** はシステムが節に自動的に割り付ける自然数である. 対象とするモジュールに含まれる最初の公理から得られた最初の節に対して 番号1が割り付けられ, 以降順次1ずつ節番号が増やされて行く. 導出節は, モジュールの公理から n 個の節が得られたとすると, 節番号 $n+1$ から割り振られる. したがって節番号の大きな導出節程, 新しく生成された節と言う事になる.

節番号の次にはかぎかっこでくくられた**<導出履歴**が表示される. これはその節がどのような推論ルール, あるいは内部処理により生成 されたかを示すものである. 導出履歴の具体的な内容については第3.15章を参照されたい. 一般にモジュールの公理から得られた節は, 導出履歴が空であるから [] と表示される. 但し, back demodulation 等によって 書き換えられる場合もあり, この場合それらに対応した導出履歴が 表示されるので, 必ずしもそうとは言えない.

最後に節の FOPL 文が表示される. リテラルが 2 個以上ある場合は, それらが論理和(|)で繋いで表示される.

3.8.2 Skolem 関数の表示形式

FOPL 文から節形式へ変換される場合, 存在限量子があった場合に Skolem 関数が自動生成される. この表示の一般形式は次の通りである：

```
定数の場合     ： #c-<N>.<ソート>
引数を持つ場合： #f-<N>.<ソート>
```

ここで**<N>** は自然数でありシステムが適当に割り付ける. また**<ソート>**は関数の値ソートの名前である.

3.8.3 変数

システムが節を処理する際には, 変数名の付け替え処理が 施される. したがって, 利用者が公理として記述した文に あった変数とは異なる変数名が与えられる. システムが生成した変数は, 次のような名前になる：

```
_V<N>
```

ここで**<N>**はシステムが自動で割り振る自然数である.

3.8.4 節の印字例

実際の節の表示例を幾つか示す. 下の例で, 1 が節番号である. 導出された節ではなく, 入力節のため導出履歴の欄は空([])となっている. 続いて節を構成する リテラルが表示されている. この場合は単一節で, リテラルは一つである. **#c-1.Account** はシステムが生成した Skolem 定数である.

```
1:[] 0 <= balance(#c-1.Account)
```

次の例(節126)も単一節の表示例であるが, 導出項であり 導出履歴欄が表示されている. この例の場合は **para-from**による paramodulation from 推論ルール に続いて, **unit-del**による unit deletion が施されたことが解る. para-from に関しては, 節 7 が paramodulator であり, これが節 2 に適用されたことが解る. 次いで 節 1 による unit deletion が 実行され, 結果としてこの節が出来た. なお, 導出履歴欄については, 第 3.15章で説明する.

```
126:[para-from:7,2,unit-del:1]
  #c-1.Int <= balance(#c-1.Account)
```

下は2つのリテラルから構成される節の表示例である. 各リテラルが論理和(|)で結合されて表示されている. **_vxx** はシステムが生成した変数である.

```
10:[] ~(_v45:Int <= _v44:Int) | 0 <= (_v44:Int - _v45:Int)
```

3.9 推論プロセスの概要

3.9.1 証明戦略 – SOS

反駁エンジンの推論機構の基本は given-clause アルゴリズムであり ([3])これは, **SOS**(Set Of Support) 戦略方式の簡単な実装の一種と見る事が できる([1]). この戦略の考え方を以下に概説する.

$A_1, \ldots A_n$ を前提，B を結論とすると, PigNose は

$$A_1, A_2, \ldots, A_n \rightarrow B$$

を証明するために反駁法を用い,

$$A_1 \wedge A_2 \cdots \wedge A_n \wedge \sim B$$

が充足不可能である事を示そうとする. その際, 全ての FOPL 文を節形式にし, resolution 原理に基づいて節を導出し て行く. その過程で空節が得られれば反駁されたと判定されるが, このとき，通 常前提は無矛盾とみなすので $A_1, A_2 \ldots, A_n$ は充足可能と仮定すると, $A_1, A_2, \ldots, A_n$ の間で，節の導出を行うのは無駄である. このアイデ アをより積極的に利用するのが SOS 戦略である.

節の集合 S の部分集合 T は $S - T$ が充足可能である時に，S の set of support と呼ばれる. PigNose の場合, 節集合 S は通常文脈モジュールで宣言された公理の集合に 対応する. set-of-support 導出 とは，どちらかの節が T に属する ような2つの節からの導出を言う. また，全ての導出が set of support 導出で あるような演繹は set-of-support 演繹 と呼ばれる.

SOS 戦略に関しては次の定理が成り立つ[1]:

> S を有限の充足不能であるような節の集合とし，T を S の部分集合 とする. もし $S-T$ が充足可能とすると，T を set of support として S から空節を導出するような set of support 演繹が存在する.

PigNose では節の集合を下のような **usable** と **sos** と呼ば れる2つに分け，導出節の生成を sos から取り出した節と usable に含まれる節 との間でのみ行うようにする．sos が上の説明の節集合 T (set of support) に対応することになる．

usable	:	推論(resolution)を行う際に使用される節の集合．
sos	:	推論には使用されない節の集合．

システムは, このようにして分割された sos 節集合から節を一つ取り出して usable に入れ, この節と元々 usable に含まれていた節との間で導出を行う．導出された節は, sos へ入れられ以降の推論に用いられる．

この方式では, 利用者が節の集合を sos と usable のどちらかに振り分ける必 要があるが, 通常は, 証明対象とするものを sos に入れれば良い．後述する自動モードでは，入力節のうち，正の節(正のリテラルのみからなる節)を sos へ入れ，それ以外の節(少なくとも一つの負のリテラルを含む節) は usable へ入れるように設定されている．この場合, 上の定義で言えば

SOS 方式による導出の例　以下の節の集合を考える．

$$
\begin{aligned}
&(1)\quad P(g(x_1,y_1),x_i,y_1)\\
&(2)\quad \sim P(x_2,h(x_2,y_2),y_2)\\
&(3)\quad \sim P(x_3,y_2,u_3)\vee P(y_3,z_3,v_3)\vee \sim P(x_3,v_3,w_3)\vee P(u_3,z_3,w_3)\\
&(4)\quad \sim P(k(x_4),x_4,k(x_4)).
\end{aligned}
$$

sos $=\{(4)\}$ とする．従って usable $=\{(1),(2),(3)\}$ である． このとき以下に示す演繹は sos を set of support とする set-of-support 演繹である：

$$
\begin{aligned}
&(5)\sim P(x_3,y_3,k(z_3))\vee P(y_3,z_3,v_3)\vee \sim P(x_3,v_3,k(z_3)) && \text{(4) と (3)}\\
&(6)\sim P(x_3,y_3,k(h(y_e,v_3))\vee \sim P(x_3,v_3,k(h(y_3,v_3))) && \text{(5) と (2)}\\
&(7)\square && \text{(6) と (1)}
\end{aligned}
$$

3.9.2　推論プロセスの主ループ

前節で反駁エンジンは sos 戦略で推論を行うことを述べたが， この方式による推論プロセスの主ループの概要は次のようになる：

while (sos が空でなく，かつ空節が導出されていない)
(1) sos から'節を一つ選'び，これを given-clause と呼ぶ．
(2) given-clause を sos から usable に 移す．
(3) '現在有効な推論ルール群'を用いて導出節を生成する．
新たに生成される節は，given-clause を一方の親として持ち，
他の親は usable に含まれる節である．

(4) 新たに生成された節に対して，‘有用性の検査’を行う．
これをパスした節は sos へ加え，そうでない節は捨てる．

end

この推論プロセスの主ループの詳細は後で説明するが, 次のような冗長性が避け られていることに注意されたい： 例えば, 節 C が 節 A と B から導出することが出来, また A と B の両方が sos に含まれているもの仮定する. もし A が given clause と して選択されると, 上の処理概要で述べた通り, これは usable へ移されて, 推 論が実行される. しかし, A と B とで C を導出することは**ない** . なぜなら, B は未だ sos にあるからである. C の導出には, B が given clause として選ばれるまで待たねばならない. さもなければ, C が2度 導出されることになってしまうからである.

sos から ‘節を一つ選’ ぶ方法 利用者がフラグによって設定する． 反駁エンジンには推論プロセスの動作を制御するために，さまざまなフラグが 用意されており， given-clause の選択方法もその一つであり，下の2つフラ グはこれに関係するものである：

sos-queue	:	sos を先入れ先出し構造(queue)と見て節の選択を行う
sos-stack	:	sos を先入れ後出し構造(stack)と見て節の選択を行う

sos-queue の場合は幅優先の探索, sos-stack の場合は深さ優先の探索に相当 する. これらのいずれのフラグも off の場合は，‘最も軽い節’ を選択する． 節の重みとは 全てのリテラルに含まれる演算子や変数の数を合計したもので ある．初期にはこ れらのフラグはどちらも off に設定されており，したがっ て節の重みによって given-clause が選択される．

重みでの given-clause の選択に関しては, 実際にはもう少し細かな制御がな される. すなわち, pick-given-ratio というパラメータが存在し, これに正 の整数値 n が設定されていた場合は, sos からの取り出し n 回目毎(最 初を含む)に, 重さを無視して, sos の先頭から節を取り出す. これを行わな いと, いつまでたっても, 初期の sos に格納されていた節からの導出節が生 成されない, という事態に陥る可能性があるからである.

given clause の選択に関係するその他のフラグについては, 第 3.11.1 節を参照されたい.

‘現在有効な推論ルール群’ 前節で述べた通り，反駁エンジンには 複数の推論ルールが用意されており，利用者がフラグによって使用したい推論 ルールを設定する．これには以下の種類がある:

フラグ名	:	推論ルール
auto	:	自動モード
auto3	:	自動モード
binary-res	:	binary resolution
hyper-res	:	hyper resolution
neg-hyper-res	:	negative hyper resolution
para-into	:	paramodulation into
para-from	:	paramodulation from

これらのフラグの初期値は全て off であり，したがって推論ルールは選択され ていない．複数のルールを同時に使用する事が可能である．自動モードでは， 入力節の集合に対して，簡単な構文的検査を行い，適当な推論ルールのセットを 選択する(第3.13節を参照).

上に示したフラグと合わせて, 推論ルールの選択に関係するフラグは, 第 3.11.3 で説明されている.

導出節の処理 導出節に対しては, 以下で示す処理が施される. 上の概要で示した '有用性のテスト' はこれに含まれるものである. 下で, ステップの番号に * 印のついているものはオプションであり, フラグの値に依存して実行するか否かが決定される.

1 変数をユニークなものにつけ替える
*2 導出節を印字する
3 demodulation を施す
*4 等式の向き付けを行う
*5 unit deletion を施す
6 同一のリテラルをマージする
*7 factor-simplification を施す
8 tautology だった場合は捨てて, 処理を終る
*9 節が重すぎる場合は捨てて, 処理を終る
*10 リテラルをソートする
11 usable あるいは sos に含まれる節によって subsume される 場合は捨てて, 処理を終る (forward subsumption).
12 索引テーブルに登録し sos に追加する
*13 追加された節を印字する
14 リテラルを一個も含まない節であれば, 反駁が発見された.
15 一個のリテラルを含む節(単一節)である場合は, usable と sos に含まれる節との間で, それと反駁するような節を探す (unit conflict).
*16 反駁が発見されたならば, 証明木を印字する.
*17 demodulator として使えるかどうかを調べる.

以上の処理は, 推論ルールによって導出された全ての導出節に対して 施され, それが終ったあとで, あらためて捨てられずに残った 全ての導出節に対して以下の処理が施される：

*18 上のステップ17で, 新たな demodulator が生成されていたならば, それらを使用して back demodulation を行う
*19 usable あるいは sos に含まれる項で, 導出節によって subsume されるような節は削除する (back subsumption).
*20 導出節の factoring を行い, 個々の factor を処理する.

3.10 フラグとパラメータの設定

フラグとパラメータは推論エンジンの実行を制御するスイッチや さまざまな制約条件を指定するためのものであり，利用者が コマンドで値を設定する (第3.6章を参照). 下はフラグとパラメータの値の設定例である. 個々のフラグやパラメータの意味 は第3.11章と第3.12章で説明する.

```
flag(binary-res, on)        -- binary resolution を有効にする
flag(back-sub, off)         -- back subsumption を無効にする
param(max-given, 100)       -- given clause の数を 100 に限定する
```

3.10.1　フラグ/パラメータ値の保存/初期化

システムが初期に起動した際には, 各フラグやパラメータは特定の既定値に初期 化される. 利用者は各人の目的に応じてこれらの値を適当に設定してシステムを 使用するわけであるが, 特定の組合せを保存しておくことが出来ると便利である. このためのコマンドが**save-option**である(第 3.6 章を参照). **save-option**で名前を付けて保存したオプション(全てのフ ラグ/パラメータとそれらの値の組み)は**option**コマンドによって再利 用することが出来る. また, オプションを全て初期値にリセットするのにも, **option**コマンドを使用する.

```
option reset               -- 全てを初期値に戻す
flag(hyper-res,on)         -- フラグやパラメータの値を
flag(back-sub, off)        -- 設定
     :
param(max-seconds, 3600)
     :
save-option option-set-1   -- 現在の設定値を option-set-1
                           -- という名前を付けて保存
     :
option = option-set-1      -- 先に作っておいたオプションを利用
```

システムを対話的に利用する場合, フラグやパラメータの値に関して混乱を生ず る恐れがある. 上で示した例のように, 一旦全てを初期値にリセットし, 次いで 各オプションの設定を行うという使い方をすると良い.

現在のフラグの設定値を知るには **list**コマンドを用いる.

```
list flag
```

また, パラメータの設定値を知るにも, 同じ**list**コマンドを 用いる.

```
list param
```

3.11　フラグ

3.11.1　given clause 選択に関するフラグ

推論の主ループの各サイクル毎に sos 節集合から取り出される節は **given clause** と呼ばれる. given clause として最も軽い節を選択する, というのが既定の方法であること は先に述べた. 節の重さとは, それに含まれるリテラルの重さの合計である. リテラルの重さとは, リテラルに含まれる項の重さの合計値であり, 項の重さは, それに含まれる演算子と変数項の数を合計したものである. given cluase の選択方法は, 以下のフラグによって制御される：

sos-queue on の時, sos を節の queue 構造とみなして, given clause を 選択する. 初期値は off.

sos-stack on の時, sos を節の stack 構造とみなして, given clause を 選択する. 初期値は off.

input-sos-first 初期値は off. もし on ならば, 初期の sos 節集合に 含まれている各節に対して, 内部的にある非常に小さな重みが設定される. 従って重みで節を選択する際, 初期状態で sos に含まれていた節が優先的に 選択されることになる.

randamize-sos 初期値は off. もし on ならば, 節の重みが 同じ節が複数あった場合, 疑似乱数を発生させて無作為に選択する.

この他, given clause の選択には, パラメータ pick-given-ratio が関係する (第3.12.3節を参照).

3.11.2 その他の主ループ動作に関するフラグ

print-given 初期値 off. on ならば given clause が選択される毎に, その節を印字する.

print-lists-at-end 初期値 off. on ならば, sos, usable 各節集合と demodulator の一覧印字を, 推論ループの終了後に行う.

3.11.3 推論ルールに関するフラグ

節の導出に使用する推論ルールを設定するフラグであり, 以下のものがある：

auto 初期値は off. on ならばシステムに導出ルールの選択や その他の制御フラグ, パラメータなどの設定を任せる. 詳しくは第3.13章を参照されたい.

auto3 初期値は off. auto フラグと同様であるが, フラグの 設定が一部異なる. 詳細は第 3.13章を参照されたい.

binary-res 初期値は off. on ならば, 新たな節の導出のため (その他に設定されている推論ルールと合わせて) binary resolution を用いる. このフラグを on にすることによって, factor および, unit-deletion フラグがそれぞれ自動的に on となる.

hyper-res 初期値は off. on ならば(他の設定されている 推論ルールと合わせて), 導出節の生成に正の hyper resolution を用いる.

neg-hyper-res 初期値は off. on ならば(他の設定されている推論ルールと合わせて), 導出節の生成に negative hyper resolution を用いる.

para-into 初期値は off. on ならば, given clause に対する paramodulation が実行される. paramodulation を用いる場合, 同値性に関する反射則($X = X$)が, 一般的には必要である. これを自動的に設定するのが, 下の universal-symmetry である.

universal-symmetry 初期値は off. on の場合, 入力節として, 同値性に関する反射則に対応する節($X = X$ に相当 するもの)を自動的に追加する.

para-from 初期値は off. on の場合, given clause を paramodulator とした(usable, および sos に含まれる 節に対しての) paramodulation が実行される. paramodulation を用いる場合, 同値性に関する反射則($X = X$)が, 一般的には必要である(上の **universal-symmetry**の説明を 参照).

demod-inf 初期値は off. on の場合, あたかも推論ルール であるかのように, given-clause に対して demodulation が実行される. このフラグが on の場合は, given-clause がコピーされ, 通常の 導出節に対してと同じ処理が施される.

prop-res 初期値は off. on の場合, 以下の条件が満足される 場合に, 命題論理的な節(変数を含まない節)に対して, binary resolution を実行する：

- フラグ **binary-res** が off
- フラグ **hyper-res** あるいは **neg-hyper-res** が on

この場合, resolution の対象となる節も変数を含まないものに 限定される.

dist-const 初期値は off. on の場合, 全ての定数は互いに異なるもの ($\neq$)として扱う. すなわち, $c_1 = c_2$ のような形のリテラルがあり, c_1 と c_2 が異なる定数項であった場合, **false** とされ, 同一の 場合は **true** として扱う. Skolem 定数に関してはこのフラグは無視 される.

3.11.4 Paramodulation に関するフラグ

Paramodulation の動作を制御するためのフラグ群であり, 以下のものがある：

para-from-left 初期値は on. on の場合, paramodulator として使用される等式 $l = r$ を, $l \to r$ と方向つけた paramodulation が行われる. para-from および para-into 両方の推論ルールの動作に有効である.

para-from-right 初期値は on. on の場合, 等式 $l = r$ を $r \to l$ と方向つけた paramodulation が 行われる. para-from および para-into 両方の推論ルールに有効である.

para-into-left 初期値は on. on の場合, paramodulation が施される対象となる正または負の等式の左辺側に 対する paramodulation が可能となる. para-from および para-into 両方の推論ルールに対して有効である.

para-into-right 初期値は on. on の場合 paramodulation が施される対象となる正または負の等式の右辺側に 対する paramodulation が可能となる. para-from および para-into 両方の推論ルールに対して有効である.

para-from-vars 初期値は off. on の場合は, 変数からの paramodulation を有効にする. このフラグを on にすると 非常に多くの paramodulation が実行される可能性がある. para-into および para-from 両方の推論ルールに対して有効なフラグである.

para-from-units-only 初期値は off. on の場合 paramodulator となり得る節は, (等式のみからなる)単一節に限られる. para-from および para-into 両方の推論ルールに対して有効である.

para-into-units-only 初期値は off. on の場合 paramodulation の対象となる節は, (等式のみからなる)単一節に限られる. para-from および para-into 両方の推論ルールに対して有効である.

para-ones-rules 初期値は off. 現在このフラグは使用されていない.

para-skip-skolem 初期値は off. 現在このフラグは使用されていない.

3.11.5 導出節の処理に関するフラグ

導出節に対して施される処理を制御するためのフラグであり, 以下のものがある：

unit-deletion 初期値 off. on の場合, 導出された節に対する unit deletion 処理が施される. unit deletion とは, 節に含まれるリテラルが, sos または usable に含まれる単一節の否定になっている 場合, それらのリテラルを除去する処理である. 例えば `p(a,X) | q(a,X)` という節の2番目のリテラルは, 単一節 `q(u,V)` によって除去される(ここで, X と V は変数であり, 他は定数とする–以下同様). しかし, 単一節 `q(u,b)` によっては除去されない. なぜなら unification によって変数 X が instantiate されるからである. この条件に合致するリテラルは全て除去され, 結果として空節が得られる 場合もある. (unit deletion は, 導出節が必ず単一節となるような場合には冗長な処理である.)

delete-identical-nested-skolem 初期値 off. on の場合, Skolem 関数がネストしているような項を持った節を除去する. 例えば, `f` が Skolem 関数であるような場合, `f(f(x))` や `f(g(f(x)))` というような形の項を含む節を除去する.

sort-literals 初期値 off. on の場合, 新たな導出節に含まれるリテラルを, 負(否定)のリテラル, 次に正のリテラルといった順に並び替える. この処理の主目的は, 節を見やすくすることであるが, 単一節でない節の subsumption 判定処理が高速化される場合もある.

for-sub 初期値 on. on の場合, 新たに導出された 節に対する処理の際に, 前向き subsumption 検査(usable あるいは sos に含 まれる節によって subsume されるかどうかの検査)が行われ, その場合は以降 の推論に無駄な節なので, これを除去する.

bak-sub 初期値 on. on の場合, 新たに保持された節を用いて, 後向き subsumption 処理(その節によって subsume されるような usable あるいは sos 内の節を除去する事)が行われる.

factor 初期値 off. on の場合, factoring 処理が 以下の2つの方式で適用される

1. 新たな導出節に対する簡単化処理：
   もし, 導出節 C が C を subsume するような factor を持っているな らば, 最も小さな factor で節を置き換える.
2. 新たな保持節に対する推論ルール:
   factoring 処理である. 他の推論ルールとは異なり, given clause には適 用されない.

3.11.6 Demodulation と等式の向きつけに関するフラグ

Demodulation の動作と, 等式の左右辺の向き付けの処理を制御するための フラグ群である.

demod-history 初期値 on. on の場合, 節が demodulate された場合, demodulator 節の id 番号を節の導出履歴の中 に保持する.

order-eq 初期値 off. on の場合, 等式の右辺が左辺より重かった場合 に左右辺を入れ換える(「重さ」の意味に関しては, 第 3.16.1 と 3.16.2 節を参照).

eq-units-both-ways 初期値 off. on の場合, 等式のみからなる単一節 に関して, 左右どちらからの向き付けも可能となる. 実際の動作は, フラグ `order-eq` の値に依存する：

- もし `order-eq` が off ならば, 等式 $\alpha = \beta$ に関して $\beta = \alpha$ が無条件に自動生成される.
- `order-eq` が on ならば, 等式の左右辺の向き付けが 出来なかった場合にのみ, $\beta = \alpha$ が生成される.

dynamic-demod 初期値 off. on の場合, 条件に叶った, 新たな保持節(処理が施されて以降の導出のために保持された 節)が, 新たな demodulator として登録される. このように, 推論過程で demodulator として追加される節を dynamic demodulator と呼ぶ. PigNose には, demodulator を利用者が指定する機能は なく, 一部例外を除いて全ての demodulator は dynamic demodulator である. 従って, demodulation 機構を使用したい場合は, この フラグを on に設定しておく必要がある.

条件については, 第 3.16.1 と 3.16.2 節を参 照. このフラグを on に設定すると, 自動的に **order-eq** も on に設 定される.

dynamic-demod-all 初期値 off. on の場合, システムは全ての新たな導出節を demodulator とすべく試みる(第 3.16.1節を参照). このフラグを on に設定すると, 自動的に **dynamic-demod** と **order-eq** が on に設定される.

dynamic-demod-lex-dep 初期値 off. on の場合 動的 demodulator (dynamic-demod フラグが on の際に, 動的に demodulator として登録される節 – 上の dynamic-demod フラグの説明を 参照 – は lex-依存, あるいは lrpo-依存 であっても良いとされる. 詳しくは第 3.16.1 と 3.16.2 節を参照.

back-demod 初期値 off. on の場合, 新たな demodulator が追加される毎に,それを用いて usable, sos, 及び現在の demodulator に対して, demodulation が行われる. このフラグが on にされると, 自動的に **order-eq** と **dynamic-demod** フラグが on に設定される.

kb 初期値 off. on の場合, システムの推論過程は Knuth-Bendix の 完備化手続きのような振舞いをする. このフラグは実際にはメタフラグであり, 下に示すようにして他のフラグを設定するものである：

```
flag(para-from,on)
flag(para-into,on)
flag(para-from-left,on)
flag(para-from-right,off)
flag(para-into-left,on)
flag(para-into-right, off)
flag(para-from-vars, on)
flag(eq-units-both-ways, on)
flag(dynamic-demod-all, on)
flag(back-demod, on)
flag(process-input, on)
flag(lrpo, on)
```

kb2 kb と同様であるが, 自動設定されるフラグのうち, para-from-vars が off にされる点が異なる.

lrpo 初期値 off. on の場合, 辞書式再帰パス順序(lexicographic recursive path ordering – lrpo) によって項の大小比較を行う. off の場合は, 重みと単純な辞書式順による比較が使用される (第3.16.2節を参照).

lex-order-vars 初期値 off. このフラグは lex-依存の demodulation と, 節に含まれる項の大小比較を行う内部手続きの動作を制御する. このフラグが on の場合, 辞書式順による項の順序付けは全順序となる. つまり, 変数は項の順序関係において最も小さく, 変数同士の順序は 名前の辞書式

順である. このフラグが off の場合は, 同じ変数かどうか という判定のみが行われる. 異なる変数同士や変数ではない項との比較は 行われない. 例えば, X と Y を変数とする. $f(X)$ と $f(Y)$ を比較すると, lex-order-vars が on の場合は比較が行われ, $f(X)$ の 方が小さいと判定される. しかしこのフラグが off の場合の比較結果は 「不明」(順序つけができない)となる.

フラグ **lrpo** が on になっている場合, lex-order-vars の値は demodulation の動作には影響を与えない(3.16.1章).

3.11.7 入力節の処理に関するフラグ

simplify-fol 初期値 on. on の場合, FOPL 文を節形式に 変換する際に, tautology の検出などを含む式の簡単化処理が施される. 簡単化処理は CNF 変換の際に Skolem 化を行った後で実施される. この際に結果が空節となる場合も有り得るが, フラグ process-input が off の場合は, システムはそれを認識しない – 反駁された(証明が見付かっ た)としない.

process-input 初期値 off. on の場合, 初期の usable および sos 節集合に含まれている節に対して, あたかもそれらが推論ルール を適用して得られた導出節であるかのような処理が施される (導出節に対する処理概要の説明が第3.9.2節にあるので これを参照されたい). 実際の導出節に対する処理と, これらの入力節 に対する処理では, 以下の違いがある：

(1) 入力節に対しては, **max-literals** および **max-weight** パラメータによる制約テストは行われない. また, **delete-identical-nested-skolem** フラグによるテストも 行われない.
(2) usable に含まれていた節は, 処理の後保持されると判定された 場合, そのまま usable に置かれる(通常の導出節は sos に追加される).
(3) 印字に関するフラグ(第3.11.8を参照) が off の場合でも, いくつかの情報が印字される.

3.11.8 印字に関するフラグ

very-verbose 初期値 off. on の場合, 詳細な処理情報が 印字される.

print-kept 初期値 on. on の場合, 新たな導出節が保持されると 判定された場合, それを印字する.

print-proofs 初期値 on. on の場合, 空節が得られた時に その導出履歴(証明木)を印字する.

print-new-demod 初期値 on. on の場合, 推論過程で動的に追加された demodulator をその都度印字する. 入力節に対する処理の場合は, このフラグの値にかかわらず, 印字される.

print-bak-demod 初期値 on. on の場合, back demodulation が実行された節をその都度印字する. 入力節が back demodulation された場合は, このフラグの値に係わらず 印字される.

print-back-sub 初期値 on. on の場合, back subsume された節をその都度印字する. 入力節が back subsume された場合は, このフラグの値にかかわらず 印字される.

quiet 初期値 off. on の場合, 全ての情報印字が抑制される.

3.11.9 その他のフラグ

control-memory 初期値 off. on の場合, 自動的に sos サイズを 元にした max-weight パラメータの再設定機能が有効となる.

order-hyper 初期値 on. on の場合, 推論ルールの **hyper-res** と **neg-hyper-res** による導出の際に, 順序付け戦略による 制約が施される. つまり, サテライト節のリテラルは, それが最大のものである (同じ節にそれより大きいリテラルが無い)ときにのみ, resolve の対象とされる. 大きさの比較は, 述語記号を辞書式順で比べる事によって行われる. (このフラグは, 正の hyper resolution – hyper-res フラグによる推論 ルールを用いる際, 全ての節が Horn 節であった場合には無意味である.)

propositional 初期値 off. このフラグが on の場合, システムは全ての節が命題論理的であると仮定して, それに最適化された処理をする.

このフラグを on にする場合, 実際に全ての節が命題論理的でなければ, 結果が正当であることは保証されない. 場合によってはシステムが clash する こともあるので, 注意されたい.

3.12 パラメータ

パラメータは正または負の整数値であり, 推論プロセスの動作を制御するものである. 本章では推論プロセスが参照するパラメータについて説明する. パラメータの有効範囲は [M ... N] という表記で規定する. これは M 以上 N 以下を 意味する. 最小値の規定で most-negative-fixnum となっているのは, Common Lisp の仕様で規定されいている最小の固定整数値, また most-positive-fixnum となっているのは同じく最大の固定整数値を意味する[8]

3.12.1 探索制約設定パラメータ

max-seconds [-1 ... most-positive-fixnum]

初期値は -1. もし $n \neq -1$ ならば, cpu 消費時間が 約 n 秒に達した時点で証明探索を中止する. 設定した時間は あくまでも目安である. システムは, ある given clause に関する 処理を全て終えた後に, このパラメータによる制約を検査する.

max-gen [-1 ... most-positive-fixnum]

初期値は -1. もし $n \neq -1$ ならば, およそ n 個の 導出節を生成した時点で, 証明探索を中止する. システムは, ある given clause に関する処理を全て終えた時点で このパラメータによる制約を検査するので, 設定した数はあくまでも 目安である.

max-kept [-1 ... most-positive-fixnum]

初期値は -1. もし $n \neq -1$ ならば, およそ n 個の節が 保持された後で, 証明探索を中止する. システムは, ある given clause に関する処理を全て終えた時点で このパラメータによる制約を検査するので, 設定した数はあくまでも 目安である.

max-given [-1 .. most-positive-fixnum]

初期値は -1. もし $n \neq -1$ ならば, 取り出した given-clause の数が n 個に達した時点で, 証明探索を中止する.

[8] Common Lisp 処理系に よってこれらの実際の値はまちまちである.

3.12.2　導出節に対する制約設定パラメータ

max-weight [-1 ... most-positive-fixnum]
　初期値は most-positive-fixnum. 生成された導出節の重みがこのパラメータで指定された値を越えた場合は, その節を捨てる. -1 は無制限を意味する.

3.12.3　その他のパラメータ

pick-given-ratio [-1 ... most-positive-fixnum]
　初期値は -1. sos から節を重みで選ぶ場合, ここで指定された数の節おきに, 重みにはよらずに, sos をリストとみなした先頭にある節を given-clause として選ぶ. -1 は指定無しを意味する.

demod-limit [-1 ... most-positive-fixnum]
　初期値は 1000. 一つのリテラルに対して一度に実行する demodulation での 書き換え回数の最大値を指定する. 書き換え回数がここで指定された数を越え た時点で, そのリテラルの demodulation は中断される. 値が -1 の場合は制 限が無い事を意味する.

max-proofs [-1 ... most-positive-fixnum]
　初期値は 1. 一度の推論プロセスで得られた空節の数が, このパラメータで指定された数に 達した場合に推論プロセスを中断する. -1 は無制限を意味する.

stats-level [0 ... 4]
　初期値は 2. 推論プロセスの最後に印字される統計情報の詳細度を指定する.

max-sos [-1 ... most-positive-fixnum]
　初期値は -1. sos に格納されている節の数がこのパラメータで指定されてい る数を越えた場合に, max-weight パラメータの自動再設定を実行する. -1 は無制限を意味する.

3.13　自動モード

本節では自動モードにおけるフラグやパラメータのセッティング, sos と usable への節の振り分けに関して説明する.

フラグ auto あるいは auto3 が on の場合, 反駁エンジンは, 入力節(文脈となっ ている CafeOBJ モジュールに含まれる公理から得られた節集合)をスキャンし, いくつかの簡単な構文的な性質を調べて推論ルールと探索戦略を自動決定する. 選択される探索戦略は(フラグ control-memory を除けば)通常 refutation complete (反駁可能な節集合であれば, 必ず反駁出来る)である. しかし, 効率の良い戦略が自動設定されるとは期待すべきではない. 自動モードでは, 多くの簡単な定理の証明が可能である. また, 証明に失敗する ような場合でも, 手動による定理証明の良い出発点を提供する.

フラグ auto/auto3 は, 他のどのフラグのセッティングよりも先に実行 されなければならない. すなわち, コマンド flag(auto,on) は他のフラグの セットコマンドよりも先にインタプリタに入力されるべきである. フラグ auto(auto3) が on にセットされた時点で, 他の依存するフラグやパラ メータが次のように自動設定される:

- process-input → on
- print-kept → off
- print-new-demod → off
- print-back-demod → off
- print-back-sub → off
- control-memory → on
- max-sos → 500
- pick-given-ratio → 4
- stats-level → 2

次いで, resolve コマンドによって反駁エンジンが起動される際に, システムは 入力節を走査し, 得られた構文的な性質の内容からどの推論ルールを使用するか を判定する. その結果適当なフラグやパラメータの設定が行われる. 入力節の走査時に調べられる構文的な性質は以下のものである：

(1) propositional か
　つまり, 全ての節が変数を持たない場合.
(2) Horn 節か
　つまり, 全ての節が高々一つの正のリテラルを持つ場合.
(3) equality があるか
　つまり, 少なくとも一つの $A = B$ あるいは $\sim(A = B)$ の形の リテラルを持つ節がある場合.
(4) equality axiom があるか
　つまり, 少なくとも一つ $A = B \to B = A$ に相当する 節がある場合.
(5) 節に含まれるリテラル個数の最大値

つぎに, これらの性質を組み合わせた簡単な 6 つのケースに分類する

(1) propositional
(2) 全ての節が単一節で equality
(3–6) {equality, Horn} の4つの組み合わせ

これらのケースに対応してどのような設定がなされるかは, システムの出力を見 る事で知る事が出来る. 一般に等式が含まれる場合は, paramodulation, demodulation が無条件で有効とされる. この場合, 細かな設定は, フラグ kb あるいは kb2 を内部的に on にすることで行う(kb および kb2 については, 第 3.11.6節を参照). auto と auto3 の違いはここのみにあり, auto の場合は kb, auto3 の場合は kb2 が on にされる. あとの動作は全く同じである.

次に入力節の sos および usable への振り分けであるが, 正の節(すべての リテラルが正の節)を sos へ, それ以外の節は usable へ入れられる.

3.14　統計情報

本節では推論プロセス実行中に収集されるさまざまな統計情報について説明する. 以下に示すのはシステムが実行中に収集される情報の一覧である.

cl-generated	推論中に生成された節の合計数.
cl-kept	導出節のうち, sos に入れられた節の合計数. フラグ process-input が on の場合は, usable や sos の初期集合に 対して行われる前処理によって, usable あるいは sos に残された節の 数も含まれる.
cl-for-sub	forward subsume されて捨てられたた節の合計数.
cl-back-sub	back subsume された捨てられた節の合計数.
cl-tautology	tautology と判断されて捨てられた節の合計数.
cl-given	given clause として sos 集合から取り出された節の合計数.
cl-wt-delete	max-weight を越えたために捨てられた節の合計数.
rewrites	demodulation による書き換え回数の総合計.
unit-deletes	unit deletion によって削除されたリテラルの合計数.
empty-clauses	推論中に導出された空節の合計数.
for-sub-sos	sos に含まれる節によって subsume され, 捨てられた 節の合計数.
new-demods	推論中に生成された demodulator の合計数.
cl-back-demod	推論中に行われた back demodulation の合計数.
sos-size	sos に含まれている節の数.
usable-size	usable に含まれている節の数.
demodulators-size	demodulator の数.
binary-res-gen	binary resolution によって導出された節の合計数.
hyper-res-gen	hyper resolution によって導出された節の合計数.
neg-hyper-res-gen	negative hyper resolution によって導出された 節の合計数.
para-into-gen	paramodulation into によって導出された節の 合計数.
para-from-gen	paramodulation from によって導出された節の 合計数.
demod-inf-gen	demodulation を行った回数.
factor-simplifications	factor simplification を行った回数.
factor-gen	factoring で生成された節の合計数.

フラグ print-stats の値が on であった場合に, 推論プロセスの終りに, それ まで収集された統計情報が印字される. 印字する統計情報の内容は, パラメータ stats-level の値に応じて内容を変え る. 一般に stats-level の値が大きい程, より詳細な情報を印字される：

- stats-level の値が 0 〜 2 の場合は以下の情報を印字する
  - cl-given
  - cl-generated
  - cl-kept
  - cl-for-sub
  - cl-back-sub
- stats-level が 3 以上の場合は全ての統計情報を印字する.

3.15　証明木とその印字

空節が得られた場合に, 空節を導出する過程に関った節を順にたどることによって, 空節の導出過程を知ることが可能である. 第 3.8章で述べられている通り, 節を印字する際には, そ の導出履歴も合わせて表示される. 従って, これらの節を識別子番号の順に並べ たリストは, 証明木(反駁木)と見る事が出来る. システムは, フラグ **print-proof** が on の場合に, 空節に出食わす毎に, このような反駁木を印字する. 下は実際の証明木印字の例である： 最後の節 163 が得られた空節である. この節を導出するのに使われた節が, 節番号の若い順に表示されている事が解る. これらの節の導出履歴を見る事により, どのような過程を経て 空節が得られたかを調べる事が出来る.

```
** PROOF _________________________________

  1:[] 0 <= balance(#c-1.Account)
  2:[back-demod:161] ~(0 <= (balance(#c-1.Account) + #c-1.Int))
  5:[] ~(0 <= _v34:Int) | balance(deposit(_v34:Int,_v33:Account))
                          = balance(_v33) + _v34
  6:[] 0 <= _v62:Int | balance(deposit(_v62:Int,_v63:Account))
                       = balance(_v63)
  10:[] ~(0 <= _v41:Int) | ~(0 <= _v42:Int) | 0 <= (_v41:Int
                                                    + _v42:Int)
  136:[para-from:6,2,unit-del:1]
    0 <= #c-1.Int
  150:[hyper:136,10,1] 0 <= (balance(#c-1.Account) + #c-1.Int)
  161:[hyper:136,5] balance(deposit(#c-1.Int,_v175:Account))
                    = balance(_v175) + #c-1.Int
  162:[back-demod:161,2] ~(0 <= (balance(#c-1.Account) + #c-1.Int))
  163:[binary:162,150]

** _____________________________________
```

3.15.1　導出履歴欄の見方

節の表示の導出履歴欄は一般的に次の形をしている：

```
<導出履歴欄> ::= <導出履歴>{,<導出履歴>}*
<導出履歴>   ::= <導出ルール>：[<節番号>{,<節番号>}*]
```

つまり <導出履歴> をコンマ記号で並べたものである. 一つの <導出履歴> は, 具体的にどのような処理によって 生成されたかを示す <導出ルール> と, オプションで <節番号> をコンマ記号で繋げたリストからなる. 例えば,

```
para-from:6,2,unit-del:1
```

では, **para-from** と **unit-del** が導出ルール に相当し, **para-from** は, 節6 と 2 が, また **unit-del** では, 節番号 1 をオプションの <節番号> として持つ. 以下各導出履歴の読み方について説明をする.

[bin-res:clause-1,clause-2]
: binary resolution(**bin-res**) によって生成された. clause-1 は given clause の節番号. clause-2 は, これと resolve した節の節番号.

[prop-res:clause-1,clause-2]
: propositional resolution(**prop-res**) によって生成された. clause-1 は given clause の節番号. clause-2 は, これと resolve した 節の節番号.

[hyper-res:clause-1,claus-2,..,clause-n]
: hyper resolution(**hyper-res**)によって生成された. clause-1 は given clause の節番号. clasuse-2 .. clause-n はこれと resolve した節 の節番号.

[neg-hyper-res:clause-1,clause-2,...,clause-n]
: negative hyper resolution(**neg-hyper-res**) によって生成された. clause-1 は given clause の節番号. clause-2 ... clause-n はこれと resolve した節の節番号.

[para-into:clause-1,clause-2]
: paramodulation into(**para-into**) によって生成された. clause-1 がオリジナルな節, clause-2 が paramodulator である.

[para-from:clause-1,clause-2]
: **para-from** によって生成された. clause-1 がオリジナルの節, clause-2 が paramodulator である.

[fsimp:]
: factor simplification(**factor-simp**)によって 生成された.

[back-demod:clause-1,claus-2,...,clause-n]
: clause-1...clause-n によって back demodulate(**back-demod**)されて生成された. この場合, もともとのオリジナルな節の内容ば破壊的に変更されている.

[demod:clause-1,clause2,...,clause-n]
: clause-1..clause-n によって demodulation が施されて生成された. **back-demod**の場合と同様, オリジナルな節の内容は破壊的に変更されている.

[copy:clause-1]
: 内部処理でclause-1の節をコピーして生成された.

[flip:]
: 等式の左右辺の入れ換えによって生成された.

[unit-del:clause-1]
: clause-1 との unit deletion(**unit-del**)に よって生成された.

[back-unit-del:clause-1]
: clause-1とのback unit deletion(**back-unit-del**) によって生成された.

3.16 項の順序付けと Demodulation

フラグ **order-eq** が on の場合には等式の左右辺の項の大小比較が行わ れ, 大きい方を左辺に持って来るよう並べ変えるという処理が行われる(これを 「等式の方向つけ」と呼ぶ). この他にもさまざまな文脈で, 項の大きさの比較 が行われる. 大小比較に際しては, 「単純辞書式順」 と 「辞書式再帰パス順序(lrpo)」 (lexicographic recursive path ordering) の2種の順序付けのうちどちらかが 用いられる(「単純辞書式順」はなんら理論的な背景を持たないことから Ad Hoc な順序付けとも呼ばれる.) フラグ **lrpo** がこれらどちらの順序つけを使用するかを決め, これが on の場合は lrpo を用い, そうでなければ ad hoc な順序つけを用いる.

3.16.1 単純辞書式順 – Ad Hoc

項の順序 (Ad Hoc)

単純な辞書式順による項の比較の場合, システムでは, 「辞書式順」と「重み -辞書式順」の2種類の順序付け方法が使われている. 利用者がこれらを選択する ことが出来るわけではなく, 順序つけを行う文脈に応じて, システムがこれらを 使い分けている.

辞書式順 これば基本的な辞書式順による記号の順序つけである. 2つの項を比較する 際に, それらを左から右へと読んで行き, 異なる演算子記号あるいは変数に 出食わした時点でストップする. これらの互いに異なる記号の大小比較の結果 が, 項の大小関係を決定する. 変数記号との比較はフラグ **lex-order-vars** の値に依存する.

lex-order-vars が on. この場合変数も比較の対象となる. 変数は記号順序で一番小さいと決められており, 変数同士はその名前の辞書 式順で大小を決める. 名前の上の辞書式順序による比較はトータル(任意の2つの名前は比較可能)なので, 辞書式順による項の順序つけもトータルである.

変数束縛によって, 相対的な項の大小関係が変化する場合があることに 注意されたい.

lex-order-vars が off (初期値). 変数は同一の変数であるかどうかだけが判定される. 従って項の順序は部分的である(大小の判定が不可能な場合がある). 変数を比較の対象とはしないので, もし, $t_1 \prec t_2$, であり, σ を変数置換とした場合, $t_1\sigma \prec t_2\sigma$ である. つまり, 変数の instantiation を行っても項の相対的な大小関係に変わりはない.

重み-辞書式順 二つの項を比較する際に, まず最初に項の重み(項に含まれるオペレータと 変数の数の合計)で比較する. 一方の項の方が重ければ, 順序においても 大きいと判定される. 両方の項の重みが同じ場合は, **lex-order-vars** が off の場合の, 辞書式順での比較と同じ方法を取る.

等式の方向付け (Ad Hoc)

フラグ**order-eq** が on で, **lrpo** が off の場合, 導出節に含ま れる等式リテラル $\alpha = \beta$ に対して α と β を比較して, 以下のような処理が施される. この処理のことを, **等式の方向つけ**と 呼ぶ.

1. どちらかの項が他の項の(真の)副項であれば, 副項が等式の右辺に 来るように並べられる.

2. もし一方の項が「重み-辞書式順」で他方より大きい場合は 大きい方が左辺に来るように並べられる.

つまり, 大きい方が常に左辺側に来るようにアレンジされる. これは lrpo を 用いた方向つけでも同じである.

Dynamic Demodulator の判定 (Ad Hoc)

フラグ**dynamic-demod**あるいは, **dynamic-demod-all**が on の 場合, フラグ**order-eq**も自動的に on になっているはずである. この場合, システムは推論の過程で導出された正の等式リテラルのみを含む単一 節を, 以降 demodulator(書き換え規則) として用いるべく試みる. もし, **process-input**が on であった場合には, usable 及び sos に含 まれている初期節集合の個々の節にたいしても同じ処理が施される. 具体的には, 等式を $\alpha = \beta$ とすると, 以下のような処理が 施される(この処理では等式が既に方向つけされているものと仮定している.) 下の説明で, $\succ$ は順序関係の大小を表現するのに用いられている. すなわち, $t \succ t'$ は t が t' より大きい事を意味する. また, $vars(t)$ は, 項 t に含まれている変数の集合, $wt(t)$ は項 t の重さである.

1. もし β が α の(真)部分項であれば, この等式を demodulator として用いる.
2. もし 「重み-辞書式順」 の意味で $\alpha \succ \beta$ であり, $vars(\alpha) \supseteq vars(\beta)$, の場合
   (a) もし **dynamic-demod-all** が on であれば, 等式を demodulator とする;
   (b) もし **dynamic-demod-all** が off であり, $wt(\beta) \leq 1$, ならば, 等式を demodulator とする.
3. もし, フラグ **dynamic-demod-lex-dep** と **dynamic-demod-all** の両方が on の場合は,
   a) α と β が変数を無視すれば同一, かつ
   b) $vars(\alpha) \supseteq vars(\beta)$,
   であれば, 等式を lex-依存の demodulator とする(次節–3.16.1 を参照).

Lex-依存の Demodulation (Ad Hoc)

含まれている変数を全て同じ X で置き換えた2つの項が, 同一となる場合, 2つの項は, 「変数を無視して同じ」と言われる. 入力節や動的 demodulator は, 等式 $\alpha = \beta$ において, α と β が「変数を無視して同じ」場合に, lex-依存の demodulator と呼ばれ る. (lex-依存の動的 demodulator の判定については, 第3.16.1節 を参照されたい).

lex-依存の demodulator は, 書き換えられた結果の項が元の項より 「辞書式順」 で小さくなる場合にのみ適用される.

3.16.2 LRPO

項の順序付け (lrpo)

辞書式再帰パス順序(lexicographic recursive path ordering) は項の比 較を行う方法もう一つの方法であり, フラグ lrpo が on の時に用いられる. 重要な理論的性質として lrpo は, termination 順序

であるという ことがある. つまり, R を demodulator の集合とし, 含まれる各 demodulator の左辺は lrpo の意味で右辺より大きいならば, demodulation (demodulator を左 辺から右辺の向きに適用する) プロセスは終了することが保証される.

lrpo による比較は, (1) 等式リテラルの方向つけ(左辺が右辺より大きく なるように置き換える事), (2) 等式が demodulator として使われ得るか, ある いは lrpo-依存の demodulator となるかを判定したり, (3) lrpo-依存の demodulator を適用できるか否かを判定するのに用いら れる. 上記以外の文脈で使用されることは無い.

オペレータ記号の順序付け – lex コマンド

lrpo を使用する場合, オペレータ記号に関する順序つけが決められてい なければならない. これは lex コマンドを用いて指定することが出来る. コマンドの構文は次の通りである：

```
<lex コマンド> ::= lex(<op1_1>,....,<op_n>)
```

<op1_1> や **<op_n>** はオペレータ記号である. 左側にある程 小さいと規定する. 特殊なオペレータ記号として ***** と **SKOLEM** の2つがシステムで予約されている：

***** lex コマンドの引数に記述されなかった残りのオペレータすべてを意 味する. これらは名前の単純辞書式順で展開され並べられる.

SKOLEM システムが生成した Skolem 関数. Skolem 関数同士は 名前の辞書式順で比較される.

例えば下のようなモジュールがあったとする：

```
module! LEX
{ [Elt]
  ops a b c : -> Elt
  op _+_ : Elt Elt -> Elt
  op _*_ : Elt Elt -> Elt
  op s : Elt -> Elt
  op _-_ : Elt Elt -> Elt
  op _/_ : Elt Elt -> Elt
}
```

ここで

```
lex(a, b, c, s, _+_, *, _*_)
```

とした場合のオペレータ順序は次のようになる：

$$\_ * \_ \succ \_/\_ \succ \_ - \_ \succ \_ + \_ \succ \mathbf{s} \succ \mathbf{c} \succ \mathbf{b} \succ \mathbf{a}$$

上の例では SKOLEM の指定が無い. * あるいは SKOLEM 指定が引数リストに無 い場合は, 以下のようにシステムが補間する：

1. * 指定がなければ, リストの最後に * を追加する.
2. SKOLEM 指定がなければ, リストの最後に SKOLEM を追加する.

つまり, これら両方が省略された時の設定は次のようになる：

```
lex(<op_1>, ..., <op_n>, *, SKOLEM)
```

また, lex コマンドの指定が無い場合のオペレータ記号の順序つけは

```
lex(*, SKOLEM)
```

とした場合と同じである. すなわち, 全てのオペレータ記号は単純辞書式順で比 較され, またシステムが生成する Skolem 関数は最も大きいと判定する. 先に述べたとおり, Skolem 関数同士の比較は辞書式順による.

lex コマンドで, * および SKOLEM というオペレータ記号が予約語とされる事に より, 利用者がこれらの名前を持ったオペレータを定義出来なくなるのでは, と懸念されるかもしれないが, 次のようにしてこれを回避出来る. いずれの場合も, 利用者の定義したオペレータを引数の数で修飾した名前を用いればよい. 例えば利用者が * という2引数のオペレータを定義している場合は, ***/2** とすることでこれを参照することが出来る. もしこれが定数であれ ば ***/0** とすれば良い. SKOLEM の場合も同様である.

lex コマンドに関する注意　lex コマンドは 'db reset' を実行する**前に**発せられていなければ ならない. 自動モードの場合は自然にこれが実現されるので問題ないが, マニュ アルモードの際には注意が必要である.

等式の方向つけ (lrpo)

もし, フラグ **order-eq** が on であり, ある等式リテラルの どちらかの引数が他方より lrpo 順で大きいならば, 大きい方の引数が左辺におかれるように並び替えられる.

Dynamic Demodulator の判定 (lrpo)

フラグ **dynamic-demod** が on の場合, システムは 全ての等式に関して, それらが demodulator として使えないかどうかを 調べる(フラグ **dynamic-demod-all** は, **lrpo** が on の場合は無視される.) 等式 $\alpha = \beta$ において, lrpo 順で $\alpha \succ \beta$ であれば, 等式は demodulator となる (この判定が行われる時点では, すでに等式の方向つけが済んでいる). もし, **dynamic-demod-lex-dep** が on であり, 等式のどちらの辺も他の辺に対して lrpo の意味で小さくなく, かつ β に含まれている変数は全て α にも出現する ものであれば, 等式は lrpo-依存の demodulator とされる.

lrpo-依存の Demodulation (lrpo)

lrpo-依存の demodulator は, 書き換えられた結果の項が 元の項より lrpo の意味で小さくなる場合にのみ 適用される.

4

詳細化検証

4.1 仕様詳細化検証システムの機能

あるモジュールMが別のモジュールM'を満足するかどうか，すなわちモジュール M' で定義された公理のすべてを M が満足するかどうかを検証する機能である．この検証の事を M' から M への詳細化検証と呼ぶ．

検証は以下の２段階で行われる：

1. M' から M へのシグニチャマッチング

   これは仕様 M が仕様 M' で定められている機能を果たすための 構文要素を備えているかどうかを検査するものである．可能な場合には M' のシグニチャから M のシグニチャへの写像 (シグニチャ射 — signature morphism)が生成される．

   シグニチャ射が一つも存在しない場合は，仕様 M が仕様 M' の機能を 果たすことは不可能であるので，詳細化検証はこの時点で失敗を報告し終了する．なお，一般的にシグニチャ射は複数存在することがある [1] ．

2. M' から M への詳細化検証

   これは1で得られたシグニチャ射による構文変換によって，仕様 M が要求 仕様 M' の機能を実際に果たすかどうかを検証するものである．M' の等式それぞ れをシグニチャ射で変換し，それが M においても成り立つかどうかを証明して いく．

   検査する M' の等式が，通常の等式(可視等式)であるか，あるいは振 舞等式であるかによって以下のように処理が異なる．

   可視等式の場合 変換された等式／公理の否定を M の仕様に加え，それから反駁が得られるかど うかを反駁エンジンを用いて検証する．反駁が得られればこの等式に関する詳細化検証は成功する．

[1] 振舞仕様におけるシグニチャは，オブジェクトにおけるメソッド及び属性に対 応すると考えてよい．シグニチャマッチングを行うには，どのメソッド／属性 がどのメソッド／属性に写像可能かを，ソート情報をもとに逐一調べていけば よい．これは一般に簡単な問題ではないが，ソート名の同じ可視ソートは同 じデータ型を意味するものと仮定することで，単純な文字列マッチングの問題 に還元され，高速な実装が可能となる．

振舞等式の場合 変換された等式について双対帰納法(coinduction)を実行する．双対帰納法 が成功すれば，この等式に関する詳細化検証は成功する(双対帰納法について は 5章を参照)．

なお現在のところ，条件付き振舞等式の条件部は全て通常の等式 (隠蔽ソートの等式であっても)であるとして処理されている． 条件付き振舞等式についてはその記述や検証の方法に関する理論的研究 が必要であり， 仕様検証システムにおける実装についても，今後の課題として残されている．

M' の全ての等式について検証が成功すれば，M' から M への詳細化検証は成功である． ただし一般に一階述語論理における定理証明は決定不能であるので， 詳細化検証が定められた計算資源(計算時間，メモリ使用量など)の上限を超え た場合には，結果を不明として報告し終了する．ただしこの場合でも，どの等 式の検証が成功しなかったかなどの情報がユーザーに提示される．

4.2 詳細化検証システムの新規コマンド

詳細化検証のための新規のコマンドは次の2つである：

シグニチャマッチングの指示 2つのモジュールを指定して, それらの間でのシグニチャマッチングを行うことを 指示する. 構文は次の通り：

```
sigmatch (<モジュール式-1>) to (<モジュール式-2>)
```

<モジュール式-1> で指定されるモジュールから, **<:モジュール式-2>** で指定されるモジュールへの, 可能なシグニチャ射を全て求め, 結果を利用者に提示する.

CafeOBJ ではシグニチャ射のことを **view** と呼ぶが, view には名前がつけられその名前で参照することが出来るようになっている. **sigmatch**では, 構成出来たview(シグニチャ射)の各々に対して適 当な名前をつけ, 利用者にはこの名前のリストを提示する. view を構成することが出来なかった場合には, 空のリストを表示する.

詳細化検証の指示 sigmatch コマンドの結果で得られた view の名前を指定して, 詳細化の検証を 行う事を指示する. 構文は次の通りである.

```
check refinement <view名>
```

検証の結果が成功であれば, “ok” と表示し, 結果が失敗あるいは不明の場合には “ng” と表示するとともに, どの等式の検証が成功しなかったかを表示する.

4.3 シグニチャマッチング

4.3.1 シグニチャマッチングの考え方

シグニチャマッチングは仕様間での構文的な対応性を検査するものであり， 仕様間でのシグニチャ射を求めることによってこれを行う.

CafeOBJ モジュールによって記述される仕様は, (S, Σ, E) の形を している. ここで (S, Σ) がシグニチャであり, S はソートの集合, Σ は引数および結果が S のソートに含まれるような オペレータの集合である. また, E はモジュールで宣言された公理の 集合であり, Σ に含まれる演算が満足しなければならない性質を記述 したものである.

シグニチャマッチングは, 二つのモジュール M と N を 与えられて, M から N に対する可能なシグニチャ射を全て計算する. M のシグニチャを (S, Σ), N のシグニチャを (S', Σ') とする. シグニチャ射とは, (S, Σ) から (S', Σ') への写像 $V : (S, \Sigma) \to (S', \Sigma')$ であり, V は二つの単射の関数

$$\begin{aligned} V : S &\to S' \\ V : \Sigma &\to \Sigma' \end{aligned}$$

から構成される. ここで, Σ に含まれる各オペレータ $f : s_1 \ldots s_n \to s$ に関して, $V(f) : V(s_1) \ldots V(s_n) \to V(s)$ が Σ' の オペレータでなければならない. 一般にこれを満足するような写像は複数あり得るので, 構成可能なシグニチャ射も一般に複数である.

可能なシグニチャ射を全て求めるのは, 一般に簡単な問題ではないが, 我々のシステムではソート名の同じ可視ソートは同じデータ型を意味するもの と仮定して問題を簡単化し，高速な計算を可能としている.

CafeOBJ ではソートの集合 S は2種のソート D と H に区分される ($S = D \cup H$). D に含まれるソートは可視ソート, V にふくまれるものは隠蔽ソートと呼ばれる. 可視ソートは通常の静的なデータ型を表現するものであり, 隠蔽ソートは内部状態を持つような動的なオブジェクトを表現するためのソートである. 同じ名前の可視ソートは同一のデータ型を意味するものとみなす，ということは 対象とする部品やシステムの仕様において，データ型が固定されている (例えばライブラリのようなものを想定する)という意味である.

4.3.2 シグニチャマッチングの例

下のような二つのモジュール, STACK と QUEUE がシステムにロード されているものとする.

```
mod* STACK(X :: TRIV) {
  *[ Stack ]*
  op empty : -> Stack
  bop top : Stack -> Elt
  bop push : Elt Stack -> Stack
  bop pop : Stack -> Stack
  vars D : Elt   var S : Stack
  eq pop(empty) = empty .
  eq top(push(D,S)) = D .
  beq pop(push(D,S)) = S .
}

mod* QUEUE(X :: TRIV) {
  *[ Queue ]*
  op empty : -> Queue
  bop front : Queue -> Elt
  bop enq : Elt Queue -> Queue
  bop deq : Queue -> Queue
  vars D E : Elt   var Q : Queue
  beq deq(enq(D,Q)) = enq(D,deq(Q)) .
  eq front(enq(E,Q)) = front(Q) .
}
```

QUEUE はキュー構造(FIFO)を, STACK はスタック構造(LIFO)を それぞれ表現したモジュールである. この状態で, **sigmatch** を実行すると次のような結果となる：

```
CafeOBJ> sigmatch (QUEUE) to (STACK)
(V#1)
CafeOBJ>
```

　この例の場合, 結果として1つの view **V#1** が得られた. これが実際にはどのような内容なのかを見るには, CafeOBJ の **show view** コマンドを用いる：

```
CafeOBJ> sh view V#1
view V#1 from QUEUE(X) to STACK(X) {sort Elt -> Elt
    hsort Queue -> Stack
    hsort ?Queue -> ?Stack
    op (Queue : -> SortId) -> (Stack : -> SortId)
    op (Elt : -> SortId) -> (Elt : -> SortId)
    op (_=*=_ : Queue Queue -> Bool) -> (_=*=_ : _ HUniversal _
                                                _ HUniversal _
                                                -> Bool)
    op (empty : -> Queue) -> (empty : -> Stack)
    bop (front : Queue -> Elt) -> (top : Stack -> Elt)
    bop (enq : Elt Queue -> Queue) -> (push : Elt Stack -> Stack)
    bop (deq : Queue -> Queue) -> (pop : Stack -> Stack)
 }
```

上の結果から Queue に関するオペレータは, Stack に関するオペレータ に対して, 次のようにマッピングされていることが分かる.

QUEUE	→	STACK
empty	→	empty
front	→	top
enq	→	push
deq	→	pop

表 4.1: QUEUEからSTACKへのマッピング

この例の場合, これ以外のマッピングは不可能である.

4.4 詳細化検証の例

本章では詳細化検証の使用例を示す.

4.4.1 QUEUE と STACK

第4.3.2 章の例で **sigmatch**コマンドによって生成された view **V#1** に 関して詳細化検証を行うと, 次のようになる：

```
CafeOBJ> check refinement V#1
no
  eq front(enq(E,Q)) = front(Q)
  beq deq(enq(D,Q)) = enq(D,deq(Q))
CafeOBJ>
```

結果は失敗であり, QUEUE に関するどの公理がSTACK において 満足されないかが印字されている. この結果は直観的にも明らかである. 例えば次のQUEUEの公理

```
eq front(enq(E,Q)) = front(Q) .
```

は, キューにある要素を追加してもキューの先頭にある要素には 変化の無い事を表現した公理である. これを view **V#1** によってSTACKモジュールに 写像すると次のようになる.

```
eq top(push(D,S)) = top(S) .
```

これはスタックに要素を追加しても先頭要素には変化が無い, ということを 言っているわけであり, したがってスタックの定義と矛盾する. 具体的には STACK の公理

```
eq top(push(D,S)) = D .
```

と相容れない.

4.4.2 モノイドと自然数上の演算の例

次に非常に単純であるが, 幾分興味深い例を示す. まず以下のモジュールを仮定する：

```
mod! TIMES-NAT {
  [ NzNat Zero < Nat ]

  op 0 : -> Zero
  op s_ : Nat -> NzNat
  op _+_ : Nat Nat -> Nat
  op _*_ : Nat Nat -> Nat

  vars M N : Nat

  eq N + s(M) = s(N + M) .
  eq N + 0 = N .
  eq 0 + N = N .
  eq 0 * N = 0 .
  eq N * 0 = 0 .
  eq N * s(M) = (N * M) + N .
}

mod* MON {
  [ Elt ]

  op null :  ->  Elt
  op _;_ : Elt Elt -> Elt {assoc idr: null}
}
```

TIME-NAT は, 自然数とその上の足し算(_+_) とかけ算(_*_) が定義されたモジュールである. モジュール MON は一般的なモノイド(単位元と2項演算をもつ代数系)を 定義したものである.

これらのモジュールを使った最初の例として, 次のような view を定義してみる：

```
  view plus from MON to TIMES-NAT {
    sort Elt -> Nat,
    op _;_ -> _+_,
    op null -> 0
  }
```

すぐに察せられるように, これはモノイドの単位元を自然数の 0, 2項演算_;_を足し算として解釈したものである. この解釈が正しいかどうかを, 仕様詳細化検証により調べると次のような 結果となる.

```
TIMES-NAT> check refinement plus
yes
```

結果はすぐに返り, 期待した通りである.

次に, モノイドの単位元 **null** を 1(**s(0)**) に, 2項演算 **_;_** をかけ算(**_*_**)にマップした view **times** を以下のように定義する：

```
view times from MON to TIMES-NAT {
  sort Elt -> Nat,
  op _;_ -> _*_,
  op null -> s(0)
}
```

この view に関して詳細化検査を行い結果が OK であれば, 自然数上のかけ算は 1 を単位元としたモノイドであると 解釈することが出来る. 上のマッピングは直観的に正しいと思われるのだが, しかしシステムは しばらく考えた後次のような結果を報告する.

```
CafeOBJ> check refinement times
no
  eq [ident12] : null ; X-ID:Elt = X-ID:Elt
```

これは TIMES-NAT において かけ算の の定義が不完全なためである. つまり_*_ が 単位元の定義 $ae = ea = a$ (e を単位元とする)を 満足するように定義されていないためである. これを修正するために, 公理

```
eq s(0) * N = N .
eq N * s(0) = N .
```

を TIMES-NAT に追加するとうまくゆく. 実際には 2 つめの公理

```
eq N * s(1) = N .
```

は, 既に宣言されている公理

```
eq N * s(M) = (N * M) + N .
```

から演繹されるので冗長である. 実際, モジュール MON において, オペレータ **_;_** の **idr:** 属性からシステムが自動生成する公理

```
eq X-ID:Elt ; null = X-ID:Elt
```

は, 先の **check** コマンドの実行結果で失敗となった公理の一覧には 表示されていない.

TIMES-NAT を上のようにして修正した後改めて view **times** を 再定義し, check コマンドを実行すると, 今度は直ちに答えが返り 成功という結果になるはずである(一般に詳細化検証で成功する場合は応答が早い).

5

モデル検査システム

5.1 モデル検査システムの機能

モデル検査システムの目的は，仕様化部品の組み合わせで構成されたシステム が，デッドロック不在やデータ整合性といった安全性を満たすことを 網羅的に検査することである． 詳細化仕様検査システムで触れた双対帰納(co-indution)も一種の安全性とみ なすことが可能であるので，モデル検査システムは振舞等式の詳細化検証にも 用いられる．

モデル検査の基本機能は，

- システムの初期状態(始状態)と検査すべき性質が 与えられ,
- 始状態から遷移可能な全ての状態について, 与えられた性質が満足されるかどうかを検査する

ことである. この場合の結果としては次の3通りが考えられる：

正常に終了 この場合，システムは与えれた性質を満たすことが保証される．

反例を発見して終了 この場合，システムの実行系列の中に与えられた性質が成立しないような反例が存在することが証明される．モデル検査システムは，その始状態からの実行 系列と反駁証明から, 反例として返す．

結果不定のまま終了 前述の詳細化検証と同じく，モデル検査は一般に決定不能であるので，事前に定められた計算資源の使用量を超過した場合には，モデル検査システムは結果 不定のまま終了する．この場合でも，探索が終了した実行系列についての情報 は提示する．

5.1.1 モデル検査の実行方式

振舞仕様におけるモデル検査については，システムの性質の記述及び検査ア ルゴリズムに関してμ計算を参考としている．μ計算系における不動点演算子 を用いることで安全性や活性以外の性質も柔軟に記述できる上に，モデル検査 アルゴリズムも不動点の繰り返し計算として定式化されため， 反駁エンジンを用いた述語計算による実装が可能となっている．

詳細を安全性のモデル検査を例にとって説明する．証明したい安全性を示す 述語をP(X:h) (X は隠蔽ソート h の変数)とすると，安全性 P のモデル検査 (後向きの)手続きは次のような述語に関する漸

化式 p_n で与えられる． ここで，隠蔽ソート上の述語が対応するシステムの状態集合の記号表現を与え ている点に注意されたい．

- $\mathbf{p_0(X)} = \neg \mathbf{P(X)}$
- $\mathbf{p_{n+1}(X)} = \mathbf{p_n(X)} \vee \mathbf{pre(p_n(X))}$
- $\neg \mathbf{p_n(init)}$ かどうかを各ステップごとに調べる

ただし init はシステムの始状態を表す(隠蔽ソート h の)定数記号であり， 状態 P(X) の前状態 pre(P(X)) は，全てのメソッド m について $(\forall \mathbf{Y})\mathbf{P(m(Y,X))}$ の選言(disjunction) をとった述語(論理 式)として定義される．ここで Y は m のアリティに適切なソートを持つ限量 変数の集合である．直観的には，$p_n(X)$ は，そこから n ステップで安全性 P(X) を破ることができるような状態の集合を表している．

上述の漸化式は収束することが保証されているいるものの，有限ステップで収 束する(ある n について $p_{n+1} = p_n$)とは限らない．したがって，アル ゴリズム化する場合には，１)有限で収束， ２) $p_n(init)$ となって反例発見， ３)どちらも不明，の３通りの結果が得られ，これは上述のモデル検査システ ムの実行結果に対応する．

前向きのモデル検査手続きについても，状態 P(X) の次状態 post(P(X))を定 義すれば同様の漸化式として定式化できる．このように，始状態(init)，状態 集合(P(X))，次状態(post(P(X)))，前状態(pre(P(X)))が定義されていれば， μ計算におけるモデル検査手続きを 反駁エンジン を用いた述語計算として行うこ とができる．いわば，この点が本モデル検査システムの核心である．

5.1.2　双対機能法の実行方式

仕様検証システムにおける振舞等式の詳細化検証に必要な双対帰納法を行うに は，状態の対(即ち関係)を状態とみなして安全性モデル検査を実行すれば良い．

例えば，振舞等式$s \sim t$を証明するには状態の対 (s,t) を始状態として 属性同値関係$=*=$(全ての属性の値が同じ)が安全性を持つこ とを示せば良い．これは以下の関係に関する漸化式として定式化でき，前述の 安全性モデル検査と同様に反駁エンジン を用いて処理することができる．

- $\mathbf{r_0(X,Y)} = \neg(\mathbf{X}=*=\mathbf{Y})$
- $\mathbf{r_{n+1}(X,Y)} = \mathbf{r_n(X,Y)} \vee \mathbf{pre(r_n(X,Y))}$
- $\neg \mathbf{r_n(s,t)}$　かどうか各ステップごとに調べる

5.2　モデル検査での新規コマンド

モデル検査では CafeOBJ に従来からある **check** コマンドを拡張した 次に示すコマンドを用いる:

```
check { safety | invariance } <述語名> [ of <文脈項> ] from <初期状態>
```

ここで，**<述語名>** は，ある隠蔽ソート H に関する述語 $P(X:H)$ の名前である． <初期状態> は，H の初期状態を表現する定数オペレータの名前である． オプションの **of <文脈項>** があった場合, 検査は**<文脈項>** で 表現されるシステム状態に対して実施される. そうで無ければ, システムが生成するソート H の変数を文脈として検査を実施する.

このコマンドの動作は以下の通りである. 検査は大きく2段階に分けて行われる. 第一段階は初期文脈に関して P が成り立つかどうかの検査であり, 第二段階は各メソッドを適用した文脈で P が成立するかどうかを 検査するループである.

初期文脈に関する検査　まず初期文脈に関する検査は以下のように行われる.

1. 初期検査対象項 t_1 を次のようにして設定する：
   - check コマンドで **<文脈項>** c が指定されているならば $t_1 = c$ とする
   - さもなければ, これを**<初期状態>**として指定されて いる項 I とする($t_1 = I$)
2. $P(t_1)$ が成立するかどうかを調べる.
   これは, 反駁エンジンを用いて, ~$P(t_1)$ が反駁出来るか 否かをしらべることによって行われる.
   - $P(t_1)$ が成立する場合 OK として第一段階を終了する.
   - ~$P(t_1)$ の反駁に失敗した場合:
     - t_1 が指定の文脈項によるものであった場合,
       a) $P(I)$ が成立するかどうかを調べる.
          * 成立するならば OK として第一段階を終了する.
          * 不成立ならば, 反例を具体的に示すため, :verb: $P(I)$ が成立するかどうかを調べる：
            · 成立すれば, この証明木が反例となる. この時点でモデル検査を終了する.
            · 不成立ならば, NG として第二段階へ行く.
     - t_1 が指定の文脈項によるもので無かった場合, 失敗として第二段階へ行く.

メソッド適用文脈での検査　上で述べた第一段階で失敗となった場合, 検査対象となった $P(t_1)$ は 反駁エンジンを用いて証明を行う際の新たな公理として追加される. これは以降で説明するメソッドを適用した文脈において証明に 失敗した文脈についても同様であり, 繰り返しの毎にその回で 失敗したものが順次追加されて行く. この新公理の集合を Ax とする.

したがって, 第一段階で失敗の場合は $Ax = P(t_1)$ である. 成功の場合は $Ax =$ (空集合) となる.

第二段階での検査は以下のようにして行われる.

1. 検査対象文脈集合 C を初期化する.
   - **<文脈項>** c が指定された場合は $C = c$ とする.
   - そうでなければ, $C = I$ とする.
2. 仮定節集合 H を空集合 に初期化する.
3. 文脈構成子集合 M を空集合に初期化する.

4. 以下を, 文脈集合 C が空になるかシステムリソースの制約条件 によって反駁エンジンが終了するまで繰り返す.
    a) C に含まれる各項 c_iについて：
        i. $C = C - c_i$ とする.
        ii. M に含まれる各文脈構成子 gen_j について, c_i と gen_j から, 検査対象項 t_ij を作る.
5. 指定された述語 P に関して, $P(c)$ が 成り立つかどうかを調べる.
6. これが成り立たなければ, 初期状態に対して $P(<$ 初期状態 $>)$ が成り立つかどうか調べる.
    a) 初期状態に対して成り立つ場合は OK とする
    b) 成り立たない場合は, $\neg P($初期状態$)$ をゴールとして 反駁を試みる事により, 反例を探す. 反例が見付かればこれを印字する. いずれの場合も, 安全性の検査に不成功として終了する.
7. H に関する全てのメソッド m_i に対して, $\forall(Y).P(m(X,Y))$ が成り立つかどうかを調べる.

5.3 モデル検査の使用例

下は銀行口座についての単純な仕様である.

```
mod! INT' {
  protecting(FOPL-CLAUSE)
  [ Int ]
  op 0 : -> Int
  op _+_ : Int Int -> Int
  op _-_ : Int Int -> Int
  pred _<=_ : Int Int

  vars M N : Int
  ax M <= M .
  ax 0 <= M & 0 <= N -> 0 <= M + N .
  ax M <= N -> 0 <= N - M .
}

mod* ACCOUNT {
  protecting(INT')
  *[ Account ]*
  op new-account : -> Account
  bop balance : Account -> Int
  bop deposit : Int Account -> Account
  bop withdraw : Int Account -> Account

  var A : Account    vars M N : Int

  eq balance(new-account) = 0 .
  ax 0 <= N -> balance(deposit(N,A)) = balance(A) + N .
  ax ~(0 <= N) -> balance(deposit(N,A)) = balance(A) .
  ax N <= balance(A) -> balance(withdraw(N,A)) = balance(A) - N .
  ax ~(N <= balance(A)) -> balance(withdraw(N,A)) = balance(A) .

}
```

モジュール INT' は整数の仕様であるが, 銀行口座の仕様を表現するために必要最低限の演算と公理のみが仕様化されている. 口座はモジュール ACCOUNT で定義されている隠れソート $Account$ によってモデル化されている. 観察子(attribute) **balance** は現在の預金高を見るものであり, メソッド **dposit** と **withdraw** は, それぞれ預金の預け入れと 引出しに相当する. 口座の初期状態は, 定数 **new-account** で表現 されており, 初期の預金残高は 0 である. これは公理

```
  eq balance(new-account) = 0 .
```

によって表現されている. その他の公理は, **deposit** と **withdraw** の動作の意味 および制約条件を表現したものである. 例えば

```
  ax N <= balance(A) -> balance(withdraw(N,A)) = balance(A) - N .
```

は, 現在の預金高が N 以上であった時に, **withdraw** が実行出来, 結果として預金高が N だけ減ることを 表現している. 一般に隠れソートを用いてこのようなシステム状態遷移を 表現する場合は, 全ての条件に対して陽に状態変化を表現してやる必要が ある. 例えば **withdraw** の場合, 預金残高が N より少ない場合 に **withdraw(N,A)** が実行されないのは上の公理で明らかで あるが, この場合でもシステム状態が「変化しない」ということを 表現する必要がある. これが, もう一つの **withdraw** に関する公理

```
  ax ~(N <= balance(A)) -> balance(withdraw(N,A)) = balance(A) .
```

である.

下は ACCOUNT モジュールでモデル検査を行うために用意した モジュール PROOF である.

```
mod* PROOF {
  protecting(ACCOUNT)

  pred P : Account .
  #define P(A:Account) ::= 0 <= balance(A) .

  op a : -> Account .
}
```

述語 P は, 口座の残高が決して 0 より小さくなる事はない, ということを表現するものである. これが実際に成立する事を, モデル検査システムを用いて調べる： 下はそのためのスクリプトである：

```
option reset
flag(auto,on)
flag(quiet,on)
param(max-proofs,1)
flag(universal-symmetry,on)
flag(print-proofs,on)
flag(print-stats,off)

open PROOF

check safety P from new-account .
```

最後の行で, モデル検査システムが起動されている. これを実行すると結果は次のようになるはずである (この例では上記のモジュール宣言ならびに実行スクリプトを **bak.cafe** という名前のファイルにいれてある.)

```
CafeOBJ> in bank
processing input : ./bank.mod
-- defining module! INT'......._...* done.
-- defining module* ACCOUNT
** system failed to prove =*= is a congruence of ACCOUNT done.
-- defining module* PROOF
-- setting flag "auto" to "on"
   dependent: flag(auto1, on)
   dependent: flag(process-input, on)
   dependent: flag(print-kept, off)
   dependent: flag(print-new-demod, off)
   dependent: flag(print-back-demod, off)
   dependent: flag(print-back-sub, off)
   dependent: flag(control-memory, on)
   dependent: param(max-sos, 500).
   dependent: param(pick-given-ratio, 4).
   dependent: param(max-seconds, 3600).
-- setting flag "quiet" to "on"
   dependent: flag(print-message, off)
-- opening module PROOF.. done.
```

```
==========
case #0-1: new-account
----------_
goal: P(new-account)*

** PROOF _________________________________

  1:[back-demod:2] ~(0 <= 0)
  2:[] balance(new-account) = 0
  7:[] _v61:Int <= _v61
  24:[back-demod:2,1] ~(0 <= 0)
  25:[binary:24,7]

** _______________________________________
** success
==========
case #1-1: deposit(_V339:Int,_hole329:Account)
----------
hypo: \A [ _V337:Int ] (\A [ _hole329:Account ] P(withdraw(_V337,
                                                          _hole329)))_
goal: \A [ _V339:Int ] (\A [ _hole329:Account ] P(_hole329)
                                                  -> P(deposit(_V339,
                                                               _hole329)))*_*

** PROOF _________________________________

  1:[] 0 <= balance(#c-1.Account)
  2:[back-demod:161] ~(0 <= (balance(#c-1.Account) + #c-1.Int))
  5:[] ~(0 <= _v34:Int) | balance(deposit(_v34:Int,_v33:Account))
                            = balance(_v33) + _v34
  6:[] 0 <= _v62:Int | balance(deposit(_v62:Int,_v63:Account))
                       = balance(_v63)
  10:[] ~(0 <= _v41:Int) | ~(0 <= _v42:Int) | 0 <= (_v41:Int
                                                      + _v42:Int)
  136:[para-from:6,2,unit-del:1]
    0 <= #c-1.Int
  150:[hyper:136,10,1] 0 <= (balance(#c-1.Account) + #c-1.Int)
  161:[hyper:136,5] balance(deposit(#c-1.Int,_v175:Account))
                    = balance(_v175) + #c-1.Int
  162:[back-demod:161,2] ~(0 <= (balance(#c-1.Account) + #c-1.Int))
  163:[binary:162,150]

** _______________________________________
** succes
```

```
==========
case #1-2: withdraw(_V337:Int,_hole329:Account)
----------_
goal: \A [ _V337:Int ] (\A [ _hole329:Account ] P(_hole329)
                                                  -> P(withdraw(_V337,
                                                                _hole329)))*_*

** PROOF _________________________________

  1:[] 0 <= balance(#c-1.Account)
  2:[back-demod:134] ~(0 <= (balance(#c-1.Account) - #c-1.Int))
  6:[] ~(_v34:Int <= balance(_v33:Account)) | balance(withdraw(_v34:Int,
                                                               _v33:Account))
                                               = balance(_v33)
                                                 - _v34
  7:[] _v58:Int <= balance(_v59:Account) | balance(withdraw(_v58:Int,
                                                            _v59:Account))
                                            = balance(_v59)
  10:[] ~(_v45:Int <= _v44:Int) | 0 <= (_v44:Int - _v45:Int)
  126:[para-from:7,2,unit-del:1]
    #c-1.Int <= balance(#c-1.Account)
  133:[hyper:126,10] 0 <= (balance(#c-1.Account) - #c-1.Int)
  134:[hyper:126,6] balance(withdraw(#c-1.Int,#c-1.Account))
                      = balance(#c-1.Account) - #c-1.Int
  135:[back-demod:134,2] ~(0 <= (balance(#c-1.Account) - #c-1.Int))
  136:[binary:135,133]

** _____________________________________


** success
** Predicate P is safe!!

CafeOBJ>
```

この例の場合は単純であり, 繰り返し探索は行われず, 深さ1レベルで 成功の元に終了している.

A

導出原理

本章では導出原理について概説する.

定理証明において仮定 A から B が結論できること すなわち $A \vdash B$ を示すためには，A を仮定してあれこれ推論規則を適用することによって B を導く． 導出原理による証明の場合は，次の同値関係

$$A \vdash B \quad \Longleftrightarrow \quad \{A, \neg B\} \vdash \square \text{ (矛盾)} \quad \Longleftrightarrow \quad \{A, \neg B\}\text{が充足不能}$$

を用いて $\{A, \neg B\}$ が充足不能であることを示すことによって 間接的に $A \vdash B$ を証明する[1].

以下このことを順を追って見てゆくことにする.

A.1 節形式と節集合

閉じた式(閉式—変数を含まない場合も含めて全ての変数が束縛されている論 理式)Aが妥当であることと $\neg A$ が充足不能であることは同じ事である． つまり $\neg A$ が充足不能であることを示すことが出来れば A が妥当で あると結論付ける事ができる． そこで式の充足不能性を確かめることを考察する.

この時スコーレム標準形(Skolem normal form)と呼ばれる，冠頭標準形に 全称限量子しか含まれない論理式の形式である．任意の一階述語論理式は 充足可能性を保存するスコーレム標準形に変換できる．この変換を スコーレム化(Skolemization)と呼ぶ．スコーレム化によって得られる式は 元の式が充足されるときそのときに限って充足される.

A.2 導出

導出(resolution)は2つの節を前提として新しい節(導出節)を 導く推論規則である．簡単のためまず命題論理の世界に範囲を 限定する．たとえば A_1, A_2, A_3 を命題として $\neg A_1 \vee A_2, \neg A_2 \vee A_3$ から $\neg A_1 \vee A_3$ を導くのが命題論理での導出の例である.

[1] ここで $\Longleftrightarrow$ は if and only if の意味.

B

反駁エンジンの動作の詳細

本章では反駁エンジンの推論過程についてより詳細な説明を行う.

B.1 推論ループの構造

反駁エンジンは, 推論終了あるいは中断の条件が検知されるまで, 指定された推論ルールにしたがって導出節を生成しつづける. この処理の構造は以下の通りである.

1. 現在の主ループ状態を「処理継続」に設定する.
2. 推論ループのための初期化処理を実行する. (第 B.2 節).
3. 初期化処理の結果, システム状態が「処理継続」ならば sos から1つ節を選択しこれを given-clause とする.
4. given-clause が存在し, かつシステム内部状態が「処理継続」 である限り, 以下の処理を繰り返す.
   a) 統計情報 cl-given に 1 を加える.
   b) フラグ print-given が on の場合, given-clause を印字する
   c) given-clause を usable の最後に追加する.
   d) given-clause からフラグで指定されている推論ルールを用いて, 節を 導出する. (第 B.3 節, 第 B.4) .
   e) 現在のシステム状態が終了条件に合致する物かどうかを調べる. (第 B.5)
   f) システムの現在状態が「処理継続」以外ならば 終了状態に関するメッセージを 出力し, ループを抜け出す.

以降で初期化から始まり，節の導出，導出後の処理，終了判定の各 プロセスについて詳細に説明する.

B.2　システムの初期化

`resolve` コマンドが発せられると反駁エンジンが起動される． その際反駁エンジンは主ループに入る前に 実行文脈の設定に続いて推論処理のための初期化処理を行う．

まず最初に反駁エンジンの実行文脈とするCafeOBJモジュールに関しての 推論の準備を行う．

1. オペレータの優先順位を設定する．
2. モジュールの公理を節形式へ変換する．

次いで反駁エンジンの主ループに入る前に，以下のような初期化処理が 実行される．

1. フラグ universal-symmetry が on の場合に，対称則(X = X)を入力節に追加する．
2. 統計情報を全て 0 に初期化する．
3. フラグ auto あるいは auto2 が on の場合にそれに応じたフラグや パラメータの自動セッティングを行う．
4. 実行文脈のCafeOBJモジュールに含まれる built-in demodulator を登 録する．
5. フラグ process-input が on の場合，sos および usable に含まれる各節に対して導出節の事前/事後処理(第 B.3.2 節および第 B.4 節を参照) を適用する．

B.3　導出節生成エンジン群の起動

システムの初期化処理に続いて フラグで指定されている推論規則に対応した 導出節生成エンジン群が順次起動され，新たな導出節が生成され， sos に追加される．

1. max-weight パラメータの調整(B.3.1節)
   a) もしフラグ control-memory が on であれば，現在の sos 節集合に 登録されている節の数を調べ，必要ならば max-weight パラメータを調整する．
2. binary resolution の実行
   a) もしフラグ binary-res が on であったならば，binary resolution を実行する (第 C.1節を参照)．
   b) binary resolution の結果得られた新たな導出節の各々について 導出節の後処理を実行する (第 B.4 節を参照 — 以下同様)．
3. hyper resolution の実行
   a) もしフラグ hyper-res が on であったならば，hyper resolution を実行する(第 C.2節を参照)．
   b) hyper resolution の結果得られた新たな導出節の各々について 導出節の後処理を実行する．
4. negative hyper resolution の実行
   a) もし，フラグ neg-hyper-res が on であったならば，nagtive hyper resolution を実行する (第 C.2 節 を参照)．
   b) negative hyper resolution の結果得られた新たな導出節の各々について 導出節の後処理を実行する．

5. paramodulation into の実行
   a) もし, フラグ para-into が on であったならば, paramodulation (into) を実行する (第 C.3節を参照).
   b) paramodulation (into) で得られた新たな導出節の各々について 導出節の後処理を実行する.
6. paramodulation from の実行
   a) もし, フラグ para-from が on であったならば, paramodulation (from) を実行する (第 C.3 節を参照).
   b) paramodulation(from) で得られた新たな導出節の各々について 導出節の後処理を実行する.

B.3.1 max-weight パラメータの調整

推論実行中の sos の爆発を避けるために用意されているパラメータが max-weight である. このパラメータ値を越えるサイズを持つ導出節は sos に登録されずに捨てられる.

さらにそのサイズが一定の目安を越えた場合に, max-weight パラメータの値を自動調整することも行う. これはフラグ control-memory が on になっている場合に実行される.

B.3.2 節に対する前処理

各入力節(フラグ process-input が on の場合)や 導出された節に関して, 例えば max-weight パラメータで制限される節の重さ等, さまざまな制約条件や論理的に冗長な節であるかどうかの検査(subsumption や tautology)を実行し, 残すべき節か捨てる物かを判別する. また dymanic demodulator の生成や空節が生成されたか否かの検査等を実行する. 具体的な処理の内容は以下の通りである：

1. 節を調べ結果として残すべきか否かを調べる (第 B.3.3 節を参照). 結果が捨てるべき節となった場合は その節を削除する.
2. 節の重みを計算する.
3. 統計情報 cl-kept の値に 1 を加える
4. フラグ print-kept が on であるか, 入力節に対する処理であった場合 clause を印字する.
5. フラグ dynamic-demod が on であり, dynamic demodulator に関する検査 (第 C.4.1 節)の結果が demodulator として ふさわしい節と判定された場合, それから新たな demodulator を作成する. この時フラグ print-new-demod が on であるかあるいは入力節に対する 処理であった場合, 作成した demodulator を印字する.
6. 空節の検査を行う. 空節であった場合,
   a) もしパラメータ max-proofs が -1 (得られる証明の数に制限を設け ない)ではなく, 導出された空節の数(統計情報 empty-clauses) の値が, パラメータ max-proofs に達していたら, システム状態を「最大証明数に達した」 として反駁エンジンを中断する(大域的脱出).

B.3.3　節に対する処理の検査内容

1. 節の変数名を付け替え，これを clause とする
2. フラグ very-verbose が on であれば clause を印字する
3. clause に demodulation を施す.
   もしフラグ very-verbose が on であり, 書き換えが一度でも実行されていれば書き換え結果の clause を 印字する.
4. 等式の左右辺の方向つけ
   フラグ order-eq が on の場合に実行する. このとき
   - フラグ lrpo が on であれば, lrpo(lexicographic recursive path ordering) による順序つけを用いる(第 3.16.2 節を参照).
   - そうでなければ, 辞書式順による等式の順序つけを行う (第 3.16.1 節を参照)
5. unit deletion 処理を施す.
   フラグ unit-deletion が on であり，かつ clause に含まれるリテラルの数が 2 以上の 場合に実行する(第 B.3.4 節を参照).
6. factor simplification の実行
   フラグ factor が on の場合に実行する (第 B.5.2 節を参照)． この時 simplification を行った回数分 統計情報 factor-simplifications を増やす.
7. tautology の検査
   節が tautology か否かを調べる(第 B.3.5 節 を参照). もしそうならば統計情報 cl-tautology を 1 増やす． このような節は推論に寄与しないためこの節を「残すべきでない」と判定する.
8. weight のテスト(導出節に対してのみ行い, 入力節に対しては 実施されない)：
   節の重さがパラメータ max-weight を 越えいた場合, この節を「残すべきでない」と判定する. この時フラグ very-verbose が on であったならば, その旨印字する.
9. forward subsumption テスト
   フラグ for-sub が on の場合に実行する(第 B.3.7 節を参照).
   結果として検査している節が他の節に subsume されるような節であった場合は, この節は冗長のため「残すべきでない」と判定し, 統計情報の cl-for-sub に 1 を加える. また, clause を subsume する節が sos 節集合に含まれているものであった場合は, 統計情報 cl-for-sub-sos に 1 を加える. さらにフラグ very-verbose が on であるか, 入力節に対する 処理であった場合は, 節が subsume された旨を印字する.

B.3.4　Unit Deletion

これは導出に使用する節を簡素化することが目的であり， 節に含まれるリテラルの数が 2 以上の場合に行われる． この処理の対象とする各リテラル l_i について, sos あるいは usable に 含まれる単一節のリテラルのうち, インスタンスが l_i の否定に等しくな るものが存在する場合に, リテラル l_i を削除する処理である.

B.3.5　Tautology 検査

```
P | ... | ~P | ...
```

という形をした節は tautology であるので, 以降の推論の役には立たない. このような節であるかどうかを判定するのが Tautology 検査である.

B.3.6　Subsumption テスト

節 C が 節 D を subsume するとは, D に含まれる全てのリテラルが C のどれかのリテラルを インスタンス化する事によって得られるような場合である． つまり C の方が D より一般的な節であるという事ができる． このような場合節 D は冗長であり, C があれば新たな導出節の生成に寄 与しない. この検査を行うのが Subsumption テストである.

B.3.7　Forward Subsumption テスト

導出された節が sos あるいは usable に含まれる節によって subsume される 場合, この節は以降の推論には役に立たない(冗長である). この検査を forward subsumption テストと呼ぶ.

B.4　節に対する後処理

節に対する後処理とは, 前処理(第 B.3.2節)を通った 節に関して行われる処理である. この処理にある節が渡される時点では, すでにその節は適切な節集合 (sos あるいは usable)に追加されいる.

前処理では節の有効性を判定する検査が既存の切に照らし合わせて種々行われ ていたのに対して, 後処理では逆に新たな導出節によって冗長になるような既存の節が あるかどうかを調べる. そのような節が存在した場合は，状況に応じて削除(back subsumption)したり, 新たに生成された demodulator によって既存の節を簡約化(back demodulation) したりする. また factoring によってさらに新たな節が生成されることもある.

具体的には以下のような処理が実行される：

1. 等式の入れ換え

   フラグ eq-units-both-ways が on かつ, 処理対象の節が単一節であり そのリテラルが equality リテラル(等式のリテラル)の場合に 以下の処理を行う：

   - フラグ order-eq が off であるか, あるいはリテラルの 等式を向きつけることが出来なかったものであったならば, 以下の処理を 行う. そうでなければなにもしない.
     a) 等式の左右辺を入れ換えたリテラルを作成し,
     b) これを唯一のリテラルとする新たな単一節を作成する
     c) 作成した節の導出履歴に copy-rule と flip-eq-rule を追加する
     d) 作成した節を, 導出節前処理 – 第 B.3.2節 – にかける.

2. back demodulation の実行
   フラグ back-demod が on かつ, clause に対応する demodulator が前処理で生成されていた場合に実行する. back demodulation は C.4.2 節を参照.
3. back subsumption のテスト
   フラグ back-sub が on の場合に行う(第 B.4.1 節を参照). clause に subsume された節の数の分だけ統計情報 cl-back-sub を増やす. このとき print-back-sub フラグが on であるか, 入力節に対する処理で ある場合には, back subsume した旨メッセージを印字する.
4. factoring
   フラグ factor が on の場合に行う.
5. back unit deletion
   フラグ back-unit-deletion が on かつ clause が単一節の場合に行う. 第 B.4.2 節を参照.

B.4.1 Back Subsumption テスト

導出された新たな節が, sos あるいは usable に含まれている節を subsume するような場合, subsume される節は冗長であるから削除することが出来る. この検査を back subsumption テストと呼ぶ.

B.4.2 Back Unit Deletion

導出節が単一節の場合, そのリテラルと符号が逆で照合可能なリテラル を削除することによって節を簡単化することが出来る. これを sos および usable に含まれる全ての節に対して実行するのが back unit deletion である.

B.5 推論終了の判定

推論途中ではパラメータの値や空節の導出を監視して推論の中断/終了の判定を 行う. まずパラメータによる 終了条件の判定は以下のように実行され, それに伴ってシステム内部状態が設 定される.

1. パラメータ max-given が -1 ではなく, 統計情報の cl-given が max-given 以上の場合, 状態を「max-given で終了」とする.
2. パラメータ max-gen が -1 ではなく, 統計情報の cl-generated が max-gen 以上の場合, 状態を「max-gen で終了」とする.
3. パラメータ max-kept が -1 ではなく, 統計情報の cl-kept が max-kept 以上の場合, 状態を「max-kept で終了」とする.
4. 上記以外なら「処理継続」とする.

空節の判定は次のようにして行う.

1. 節に含まれるリテラルの数が 0 の場合.
   これは空節であり, 以下の処理を実行する.

a) フラグ print-message が on であれば空節が導出された旨印字する.
b) 統計情報 cl-kept を 1 増やす.
c) 節が含まれている節集合からこの節を削除する.
d) 統計情報 empty-clauses を 1 増やす.
e) フラグ print-proofs が on であれば証明木を印字する.

2. リテラルの数が一つの場合(単一節)
unit conflict があるかどうかを調べる(第 B.5.1 節を参照). その結果 unit conflict となった場合, 以下の処理を行う.
a) print-message フラグが on の場合 unit confict となった旨印字する.
b) フラグ print-proofs が on の場合証明木を印字する.
c) 節の含まれている節集合からこの節を削除する.

B.5.1 Unit Conflict 検査

与えられた単一節を反駁するような, sos あるいは usable に 含まれる単一節の全てを探し, もしあればそれぞれについて反駁を 実行して空節を作り出す処理である.

B.5.2 Factoring 処理

節 C に含まれる 2つ以上の(同じ符号の)リテラルに，mgu(most general unifier) σ があるとしたとき，σC (C の各リテラルのアトムに, σ を適用した結果) を C の factor と呼ぶ.

例えば， $C = \underline{P(x)} \vee \underline{P(f(y))} \vee \sim Q(x)$ としたとき， (下線を引いた)最初と2番目のリテラルは, mgu $\sigma = \{f(y)/x\}$ を持つ． したがって，$\sigma C = P(f(y)) \vee \sim Q(f(y))$ は C の factor である.

Factor Simplification 処理

factor simplification は, ある節の factor が元の節を subsume する ものかどうかを調べ, そうであれば元の節のリテラルを factor の リテラルで置き換える. このようにして clause を簡単化して, 冗長な リテラルを削除して行く処理である.

C

推論ルール概説

本章では各種の節導出方式について簡単に説明する.

C.1 Binary Resolution

binary resolution はリテラルとリテラル 1 対 1 の通常の resolution ルールである. resolution のスキームは図 C.1 に示す通りである：

$$\begin{array}{ll} \textbf{clause-1:} & \sim L', M_1, \cdots, M_m \\ \textbf{clause-2:} & L, K_1, \cdots, K_k \\ \hline \textbf{binary resolvent:} & \sigma K_1, \cdots, \sigma K_k, \sigma M_1, \cdots, \sigma M_m \end{array} \quad \sigma L L' m g u$$

図 C.1: binary resolution のスキーム

図に示したスキームで, 二つの親節は同一の変数項を共通して持っていてはならない. この条件は, 生成節の変数は常にユニークになるように変数のつけ替えが行われて いるため, 自動的に満足されている.

C.2 Hyper Resolution

ある節から導かれる導出節を考えた場合, それを得るのに 異なった導出ステップがあり得る. 従って, 同じ導出節が複数回生成される場合があり得る. これは望ましくないので, この複数のステップをまとめて 一度の resolution によって導出しようとするのが hyper resolution である.

反駁エンジンでは，positive hyper resolution と negative hyper resolution の2種の hyper resolution 機能を提供しているが，これらはいずれも 一種の semantic resolution であり，PI-resolution と呼ばれるより一般的な resolution rule の特殊形である.

ここではまず最初に，この semantic resolution の考えかたについて説明し([1])， ついで，hyper resolution のスキームについて説明する．

C.2.1 Semantic Resolution の考え方

[1]

節の集合 S を， $S = \{\sim P \vee \sim Q \vee R, P \vee R, Q \vee R, \sim R\}$ とする．この節集合の充足不能性を示すのに，通常の binary-resolution を 使うと，以下に示すような演繹過程となる：

(1)	$\sim P \vee \sim Q \vee R$	S の要素
(2)	$P \vee R$	S の要素
(3)	$Q \vee R$	S の要素
(4)	$\sim R$	S の要素
(5)	$\sim Q \vee R$	(1) と (2)から
(6)	$\sim P \vee R$	(1) と (3)から
(7)	$\sim P \vee \sim Q$	(1) と (4) から
(8)	P	(2) と (4) から
(9)	Q	(3) と (4) から
(10)	$\sim Q \vee R$	(1) と (8)から
(11)	$\sim P \vee R$	(1) と (9)から
(12)	R	(2) と (6) から
(13)	$\sim Q$	(4) と (5) から
(14)	$\sim P$	(4) と (6) から
(15)	$\square$	(4) と (12) から

導出節のうち，実際に証明に使われているのは (6) と (12) だけである． その他の導出節は全て冗長である．この様な冗長な節の導出を避けるのが 目的となる．

節を適当な2つのグループ S_1 と S_2 に分けることが出来たとする. また, おなじグループに含まれる節同士の間では resolution を行わないとする. このようにする事で, 導出される節の数が減ることはあきらかである.

semantic resolution では, 節のグループ分けに**解釈**(interpretation) を用いる．上の例で, (2) と (3) は, 解釈 $I = \{\sim P, \sim Q, \sim R\}$ では false である. 一方 (1) と (4) は満足される. このようにすることで, S を解釈で true になるものとそうでないものとに分類することが出来る. 反駁エンジンは, 節の充足不可能な節集合を扱うものであるから, どのような解釈を採用したとしても, それが全ての節を満足したり否定する ようなことは無い. したがって, どのような解釈によっても節集合を 2つの集合に分類することが可能である. 以下では, 先の例の節集合 S が上で述べた解釈 I によって, 二つのグループ $S_1 = \{(2), (3)\}$, $S_2 = \{(1), (4)\}$ に分類されているものとする. このように分類しても, (2) と (3) は (4) と resolve することが出来る. 以下で述べるように, この resolution についても避けることが可能である.

上のような resolution を回避するため, 述語記号の間の順序付けを使うことが 出来る. たとえば, S に含まれる述語記号に

$$P > Q > R$$

という順序を付けるものと仮定する. この順序付けのもとで, 2つの節(片方は S_1 の節, もう片方は S_2 の節) を resolve する場合, resolve されるリテラルで S_1 に属する節のものは 最大の述語記号をもっていなければならない, という制約条件を付ける. この制約のもとでは, R は (2) および (3) で最大の述語記号ではないため, (2) を (4) と resolve することは出来ないし, また, (3) と (4) に関しても同様である.

次に semantic resolution で重要となる概念である clash について説明する. まず, 先の例の節について見てみる.

(1) $\sim P \vee \sim Q \vee R$
(2) $P \vee R$
(3) $Q \vee R$
(4) $\sim R$

解釈 I を $I = \{\sim P, \sim Q, \sim R\}$ とし, 述語記号の順序を $P > Q > R$ とする. 上で議論した2つの制約を適用すると, 次の2つの 節のみが導出出来る：

(5) $\sim Q \vee R$ (1)と(2)から
(6) $\sim P \vee R$ (1)と(3)から

これらの両方とも I によって満足される. したがってこれらを S_2 へ入れ, それらを S_1 に含まれる節と resolve する. (5) は (3) と (6) は (2) と resolve 出来る. 両方とも同じ結果 R になる.

(7) R (3)と(5)から
(8) R (2)と(6)から

節 R は I では false なので, それを S_1 へ入れ, これと S_2 に含まれる 節との間で resolve を行う. $\sim R$ が S_2 にあるので, 空節 □ を得る.

上の二つの R の導出過程を見ると, ともに節 (1), (2) と (3) が使われている. 異なるのはそれらが使われる順序である. どちらか一つだけが必要であり, 片方は 冗長であり, 極めて無駄である. この冗長性を無くすために, clash という概念を 導入する. clash の考え方は, (1), (2), (3) の3つの節から, 中間的な節 (5) や (6) を経由せずに, 直接的に R を導出するというものである. この例では, 集合 $\{(1), (2), (3)\}$ を clash と呼ぶ. clash の詳細に ついては, 次のパラグラフで説明する.

C.2.2 PI-resolution の定義

I を解釈とし, P を述語記号の順序付けとする. 節の有限集合 $\{E_1, \ldots, E_q, N\}$, $q \geq 1$ は 次の条件を満足する時, semantic clash と呼ばれる：

1. $E_1, \ldots, E_n$ は I で false

2. $R_1 = N$ とする. 各 $i = 1, \ldots q$ について, R_i と E_i の resolvent R_{i+1} が存在する.
3. resolve される E_i のリテラルは, $E_i, i = 1, \ldots q$ の うちで最も大きな述語記号を持っている.
4. R_{q+1} は I において false

R_{q+1} を (PI-clash $\{E_1, \ldots, E_q, N\}$ の) PI-resolvent と呼ぶ. また, $E_1, \ldots, E_n$ を **electrons**, N を **nucleus** と呼ぶ.

例えば,

$$E_1 = A_1 \vee A_3, \qquad E_2 = A_2 \vee A_3, \qquad N = \sim A_1 \vee \sim A_2 \vee A_3$$

とする. 解釈 I を $I = \{\sim A_1, \sim A_2, \sim A_3\}$ とし, 順序 P を $A_1 > A_2 > A_3$ とする. この場合, $\{E_1, E_2, N\}$ は PI-clash である. この clash の PI-resolvent は, A_3 である(A_3 は I で false であることに注意).

C.2.3 Negative Hyper Resolution

negative hyper resolution は, 上で述べた PI-resolution で 解釈 I に否定記号が含まれていない場合である.

negative hyper resolution のスキームを図C.2に示す.

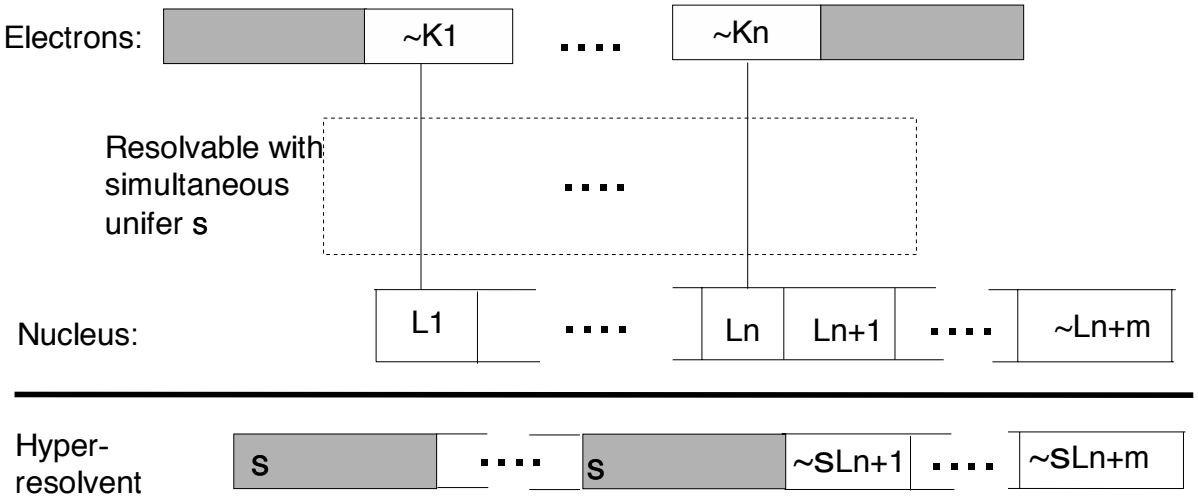

図 C.2: hyper resolution のスキーム

図で nucleus とあるのは, 少なくとも一つの正のリテラルをもつ節である. 反駁可能な節集合にはこのような節が必ず存在する. nucleus の個々の正のリテラルにつき, 負のリテラルのみを持つ節 electron が必要である. やはり, 反駁可能な節集合にはこのような節が必ず存在する. nucleus は全ての electron と「同時に」resolve されなければならない. 結果として負のリテラルのみからなる節が, 導出節として得られる(図の hyper-resolvent). この導出節は, 以降の hyper-resolution ステップで electron として使用することが出来る.

我々の反駁エンジンでは，フラグ neg-hyper-res が on の時に 使用される negative-hyper-resolution に相当する．

C.2.4 Positive Hyper Resolution

フラグ hyper-res が on の時に使用されるのは，これと dual な関係にある，positive hyper resolution である．これは，上で述べたスキームで，リテラルの 正/負 を逆にする事で得られ， PI-resolution で，解釈 I に含まれる各リテラルが否定記号を含む場合に 相当する．

否定の形で与えられる結論は，通常負のリテラルのみを含むことが多いため，negative hyper-resolution の electron として用いる事が出来るので，結論から仮定へ向かっての，後向き推論に適していると言える．一方 positive hyper-resolution は，仮定から結論へ向けての，前向きの 推論が可能である．

C.3 Paramodulation

paramodulation は等号を扱うための推論ルールである．図 C.3 に paramodulation のスキームを示す．図で，$l = r$ は **paramodulator** (正の等式リテラル)，$f(t)$ は t を副項に持つようなリテラル である．$f(\sigma r)$ は，その t を $\sigma(r)$ で置き換えた項を 表すものとする．

$$\frac{\begin{array}{ll} l = r, L_1, \cdots, L_n & \\ f(t), M_1, \cdots, M_m & \sigma ltmgu \end{array}}{\textbf{paramodulant:} \quad f(\sigma r), \sigma L_1, \cdots, \sigma L_n, \sigma M_1, \cdots, \sigma M_m}$$

図 C.3: paramodulation のスキーム

図で，$sigma$ は，リテラル $f(t)$ の副項 t と paramdulator の 左辺 l との mgu(most general unifier) である ($\sigma l = t$)．paramodulant は t を paramodulator の右辺に $sigma$ を適用した もの($f(\sigma r)$) で置き換えることによって得られる．

C.4 Demodulation

demodulation は，導出された節に適用され，それに含まれるリテラルのアトムを 簡約化する． すなわち，$l \to r$ の形をした demodulator があり，あるリテラル l が項 t を副項として含む $P[t]$ の形をしているものとする． このとき，$\sigma l = t$ となるような，変数置換があったときに， l を，$P[\sigma r]$ に書き換える．

demodulator として，$L = R$ という等式のリテラルのみからなる節が用いられる． これを，項の書き換え規則として用いるためには， 等式の方向付けが必要である． つまり，何らかの順序関係に

よって，等式の左右辺を方向付ける必要が ある．反駁エンジンでは，この順序の判定に，単純な辞書式順と lrpo の 2種 を用意する．

C.4.1　実行時のdemodulator判定

フラグ dynamic-demod が on の場合, 反駁エンジンは, 全ての equality ($\alpha = \beta$)を demodulator として使えるか どうかを判定する.

フラグ dynamic-demod あるいは, dynamic-demod-all が on になっている 状況では, 必ずフラグ order-eq が on になっているはずである. この時フラグ lrpo が on の場合には, 等式の向きつけの判定として LRPO を, そうでない場合は単純な重みと辞書式順による比較が行われる. この結果等式の向きつけが行われるが, その 節が demodulator として使えるかどうかは, 次のような処理で 判定される. なお検査の対象とする節は, 正の equality かつ単一節である.

1. 節のリテラル $l(\alpha = beta)$ に関して以下の判定を実行する：
   フラグ lrpo が off の場合　a) β が α の副項であれば :normal とする
      b) 辞書式順の意味で $\alpha > \beta$ かつ $vars(\alpha) \supseteq vars(\beta)$ ならば,
         i. フラグ dynamic-demod-all が on であれば OK とする
         ii. dynamic-demod-all が off で $wt(\beta) \leq 1$ ならば OK とする
         iii. それ以外の場合は NG とする
      c) フラグ dynamic-demod-lex-dep と dynamic-demod-all の 両方が on のとき, α と β が変数を除外すれば, 構文的に同一の項である場合に ORDER-DEP とする.
      d) それ以外の場合は NG とする
   フラグ lrpo が on の場合　a) l の等式の向きつけが正しく行われている場合, OK とする.
      b) フラグ dynamic-demod-lex-op が on の場合, $vars(\alpha) \supseteq vars(\beta)$ ORDER-DEP とする.
      c) それ以外の場合は NG とする

上の処理で, $vars(t)$ は, 項 t に出現する変数の集合を意味する.

C.4.2　Back Demdulation の実行

back demodulation とは，導出された節が正の単一の equality 節 (一つの $\alpha = \beta$ の形のリテラルのみからなる節) で あった場合に，それを demodulator として用いて，usable および sos に含まれる全ての節に関して demodulation を実行するものである.

導出節が demodulator として適切なものかどうかの判定は， 導出節に対する前処理(第B.3.2節 を参照)で行われている.

参考文献

[1] Chang, C. and Lee. R.C., Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973

[2] Joseph Goguen and Grant Malcolm, “A Hidden Agenda”, in Theoretical Computer Science, Vol.245 No.1, 2000, pp.55–101

[3] William McCune, “Otter3.0 Reference Manual and Guide”, Technical Report ANL-94/6, Argonne National Laboratory, 1994, available at **http://www-unix.mcs.anl.gov/AR/otter/**

[4] A.T.Nakagawa, T. Sawada, and K. Futatsugi, “CafeOBJ Manual”, avaiable at **ftp://ftp.sra.co.jp/lang/CafeOBJ/Manual/**

[5] Răzvan Diaconescu and Kokichi Futatsugi, CafeOBJ Report. World Scientific, 1998