JAIST Repository
https://dspace.jaist.ac.jp/

Title	COE Research Monograph Series, Vol. 1 : オブジェクト指向理論によるファイアーウォールサーバの検証
Author(s)	矢竹，健朗
Citation	
Issue Date	2006-07
Type	Book
Text version	publisher
URL	http://hdl.handle.net/10119/3793
Rights	
Description	The original publication is available at JAIST Press http://www.jaist.ac.jp/library/jaist-press/index.html

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

オブジェクト指向理論による
ファイアウォールサーバの検証

情報科学研究科
矢竹 健朗

はじめに

本書のテーマは，定理証明によるオブジェクト指向分析モデルの検証である．定理証明は，モデル検査と並び，近年，情報産業界でも注目されつつあるシステム検証技術の 1 つである．我々は，定理証明器 HOL においてオブジェクト指向モデルを検証するためのツール ObjectLogic を実装した．本書では，ObjectLogic の紹介，及び，ファイアウォールサーバの検証事例について述べる．

情報システムが社会インフラとなっている現在，情報システムの欠陥が社会に甚大な被害をもたらすようになっている．最近では，銀行システムや証券取引システムのダウンが記憶に新しい．情報インフラの脆弱性は，単に企業の損失だけでなく，国家の信頼性の低下をも招いてしまう．我々が安全に，安心して暮らせる社会を構築するためには，情報システムが正しくつくられ，要求仕様通りの動作をすることを保証する必要がある．

これまで，システムの正しさの検証は主にテストによって行われていた．テストとは，システムに対して具体的な入力を与え，予期される結果が出力されるかどうかを確認する手法である．テストの問題点はそのカバレッジを 100%にすることが不可能な点である．つまり，システムの入力値は一般に無限であり，それらのケースを一つづつ確認していくことは理論的に不可能である．確認されなかったケースについてはシステムの穴となりうる．テストのもう 1 つの問題点として，ある程度システムが完成してからでないと行えないことが挙げられる．実装段階で発見されたバグの修復には大きなトラックバックが発生する．最悪の場合，一からやり直しということもありうる．また，発見されない場合は，製品に欠陥をもたらし，回収や事故による大きな金銭的損害，あるいは人的損害が生じてしまう．

これを克服する手法として最近では，厳密な数学に基づく検証法が注目されつつある．代表的なものは，モデル検査，定理証明である．これらはいずれも隙間のない，網羅的な検証が可能である．モデル検査ではシステムの振る舞いを有限状態機械によって表し，探索アルゴリズムによりシステムのとりうる全状態の検査を行うことが可能である．定理証明ではシステムに出現するデータ型に対し帰納法を適用することにより，すべての値に関して求める性質が成立することを証明可能である．これらの検証法を分析段階において適用することにより，欠陥を早期発見することが可能である．

我々はこれまでに，定理証明に基づく検証法として，高階述語論理に基づくオブジェクト指向分析モデルの検証ツール ObjectLogic を実装した．ObjectLogic では，定理証明器 HOL[2] に実装されたオブジェクト指向理論において，モデルが要求仕様を満たすことを検証することが可能である．また，関数型言語 moscow ML[3] においてモデルのシミュレー

ション実行も可能である．コストの高い定理証明に先立ち，比較的コストの小さいシミュレーション実行によってできる限りバグを取り除くことにより，検証プロセス全体でのコストを削減することができる．実際の製品の論理的な振る舞いを実行可能モデルとして構築することは，特に大規模システムの場合，システム全体の振る舞いを把握するために有用である．

本書の構成

本書は 3 部構成となっている．

第 I 部では，オブジェクト指向モデル検証ツール ObjectLogic について述べる．第 1 章では，ObjectLogic の概要を簡単な例題を通して説明する．第 2 章では，HOL におけるオブジェクト指向理論の定義を述べる．第 3 章では，オブジェクト指向理論の実装を述べる．第 4 章では，UML[1] でモデル化された図書館システムの検証例を述べる．第 5 章では，関連研究との比較を行う．

第　部では，実践的な例題として，ファイアウォールサーバの検証について述べる．第 6 章では，ファイアウォールサーバの仕様を述べる．第 7 章では，UML のクラス図，シーケンス図を用いたモデル化を述べる．第 8 章では，ML におけるシミュレーション実行を様子を述べる．第 9 章では，HOL における定理証明について述べる．

第　部は付録集となっている．

関連サイト

ObjectLogic に関する情報は，

http://www.jaist.ac.jp/~k-yatake/hol/objlog

で公開している．

目 次

第 I 部　オブジェクト指向モデル検証ツール ObjectLogic **13**

第 1 章　ObjectLogic の概要 **15**
1.1 クラスモデル 16
1.2 ML におけるシミュレーション実行 16
1.2.1 プリミティブ演算子 17
1.2.2 メソッドの定義 23
1.3 HOL における定理証明 30
1.3.1 型，演算子，公理 30
1.3.2 メソッドの定義 33
1.3.3 証明例 34

第 2 章　オブジェクト指向理論 **47**
2.1 クラスモデル 47
2.2 型，定数 48
2.3 演算子 48
2.4 公理 49

第 3 章　オブジェクト指向理論の実装 **59**
3.1 背景 59
3.2 概要 60
3.3 ヒープメモリの実装 62
3.3.1 サブヒープのデータ構造 62
3.3.2 オブジェクト参照の表現 62
3.3.3 ヒープメモリのデータ構造 63
3.3.4 ヒープメモリにおけるオブジェクト指向演算子の表現 66
3.3.5 ストア型の生成 68
3.3.6 演算子の定義 70
3.3.7 公理の導出 71

第 4 章　UML の検証 **73**
4.1 図書館システム 73

4.1.1 クラス図 . . . 73
4.1.2 シーケンス図 . . . 75
4.2 HOL における定理証明 . . . 77
4.2.1 メソッドの定義 . . . 78
4.2.2 メソッド事前事後条件の証明 . . . 78
4.2.3 不変表明の証明 . . . 83

第 5 章 関連研究 91
5.1 Deep embedding vs shallow embedding . . . 91
5.2 ヒープメモリに基づくオブジェクト指向理論 . . . 92
5.3 Extensible record による構造的サブタイピング . . . 92
5.4 Z との比較 . . . 93

第 II 部 ファイアウォールサーバのモデル化と検証 95

第 6 章 ファイアウォールサーバの概要 97
6.1 パケットフィルタリングシステムの仕様 . . . 97
6.2 パケットフィルタリングの例 . . . 99
6.2.1 アウトバウンド送信の例 . . . 99
6.2.2 インバウンド送信の例 . . . 100

第 7 章 UML によるモデル化 103
7.1 クラス図 . . . 103
7.2 シーケンス図 . . . 108
7.2.1 pfm.filterOut() . . . 108
7.2.2 pfm.filterIn() . . . 110
7.2.3 pfm.connectionExists() . . . 113
7.2.4 pfm.isPhysicallyConnectable() . . . 113
7.2.5 contable.getConnection() . . . 114
7.2.6 pfm.srcnat() . . . 114
7.2.7 nattable.srcnat() . . . 114
7.2.8 nattable.addNatrule() . . . 117
7.2.9 pfm.updateConnection() . . . 118
7.2.10 pfm.addConnection() . . . 118
7.2.11 pfm.isValidPacket() . . . 119
7.2.12 pfm.incSec() . . . 120
7.2.13 contable.decTimer() . . . 120

第 8 章 ML におけるシミュレーション実行 123

第 9 章　HOL における定理証明 **131**
9.1 **パケット通過許可条件の証明** 131
9.2 **アドレス変換正当性の証明** 136
9.3 **接続表とアドレス変換表の一貫性証明** (1) 144
9.4 **接続表とアドレス変換表の一貫性証明** (2) 157

第 III 部　付録集 **161**

付 録 A パケットフィルタリングシステムの HOL コード **163**
A.1 **クラスモデル** 163
A.2 **定数，ユーティリティ関数の定義** 164
A.3 packet **クラスのメソッド** 166
A.4 natrule **クラスのメソッド** 167
A.5 nattable **クラスのメソッド** 169
A.6 frule **クラスのメソッド** 174
A.7 doscounter **クラスのメソッド** 176
A.8 connection **クラスのメソッド** 177
A.9 contable **クラスのメソッド** 178
A.10 pfm **クラスのメソッド** 182

付 録 B 公理・演算子対応表 **195**

付 録 C HOL における論理記号の表記 **197**

参考文献 **199**

図 目 次

1.1 objLib の機能 15
1.2 figure.cm 16
1.3 クラスモデルの例 16
1.4 figure.sig (1) 18
1.5 figure.sig (2) 19
1.6 rect クラスのメソッド symX(), symY(), symO() 28
1.7 動的束縛の実現 29
1.8 UP_DOWN 33
1.9 UP_11 33
1.10 IS_CAST 34
1.11 HOL におけるメソッドの定義 35

3.1 プログラムと分析モデルの違い 60
3.2 ヒープメモリの表現 61
3.3 サブヒープ上の演算子 63
3.4 ヒープメモリのデータ構造 64
3.5 ヒープメモリ上の演算子 65
3.6 ヒープメモリからストアへの抽象化 69

4.1 図書館システムのクラスモデル 74
4.2 library::lend() 75
4.3 library::checkLend() 76
4.4 library::getCustomer() 77
4.5 HOL におけるメソッド lend() の定義 (1) 79
4.6 HOL におけるメソッド lend() の定義 (2) 80
4.7 利用者の貸出総数の変化 86
4.8 貸出中の物品総数の変化 88

6.1 アウトバウンドフィルタリングの例 (1) 100
6.2 アウトバウンドフィルタリングの例 (2) 101
6.3 インバウンドフィルタリングの例 102

7.1 パケットフィルタリングシステムのクラス図 105
7.2 filterOut(), filterIn() に関わるクラス図 106
7.3 incSec() に関わるクラス図 106
7.4 コンフィギュレーションの設定に関わるクラス図 107
7.5 pfm::filterOut() 109
7.6 pfm::filterIn() 111
7.7 filterOut(), filterIn() の制御フロー 112
7.8 pfm::connectionExists() 113
7.9 pfm::isPhysicallyConnectable() 114
7.10 contable::getConnection() 115
7.11 pfm::srcnat() 115
7.12 nattable::srcnat() 116
7.13 nattable::addNatrule() 117
7.14 pfm::updateConnection() 118
7.15 pfm::addConnection() 119
7.16 pfm::isValidPacket() 121
7.17 pfm::incSec() 122
7.18 contable::decTimer() 122

9.1 検証対象の OCL による記述 132
9.2 アドレス変換の正当性 136
9.3 アウトバウンドパケット処理に関する 3 つの場合 141
9.4 接続表とアドレス変換表の一貫性 145
9.5 前提不変表明の OCL による記述 148
9.6 pfm_filterOut による接続表とアドレス変換表の変化 (1) 152
9.7 pfm_filterOut による接続表とアドレス変換表の変化 (2) 154
9.8 pfm_filterOut による接続表とアドレス変換表の変化 (3) 156
9.9 pfm_incSec1 による接続表とアドレス変換表の変化 159

表 目 次

B.1 **公理・演算子対応表** (1) 195
B.2 **公理・演算子対応表** (2) 196

C.1 **論理記号の表記** 197

第I部

オブジェクト指向モデル検証ツール
ObjectLogic

第1章　ObjectLogicの概要

本章では，オブジェクト指向モデル検証ツール ObjectLogic の概要を述べる．ObjectLogic は，objLib という名前の HOL ライブラリ（HOL のメタ言語 moscow ML のストラクチャ）として実装している．このツールを用いれば，モデルを ML でシミュレーション実行したり，HOL で定理証明を行うことが可能になる．

シミュレーション，定理証明を行うためには，対象システムのクラスモデルからシミュレータ，オブジェクト指向理論を生成する必要がある．それを行うのが objLib が提供する関数 mk_theory_script, mk_simurator である（図 1.2）．

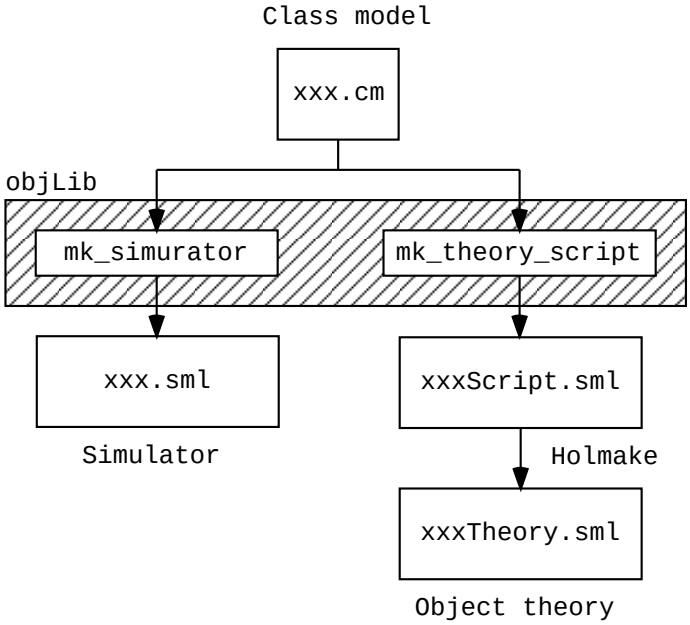

図 1.1: objLib の機能

まず，両関数の共通の入力となるクラスモデルについて述べる．次に，シミュレータを生成し，モデルを実行する方法について述べる．最後に，オブジェクト指向理論を生成し，定理証明を行う方法について述べる．

1.1　クラスモデル

クラスモデルは，クラスや属性，継承関係など，システムの静的構造を定義したモデルであり，ファイルとして作成する．図 1.2 はクラスモデル figure.cm の内容である．

```
class fig
attr x  : int | 0
attr y  : int | 0

class rect extends fig
attr w  : int | 0
attr h  : int | 0

class crect extends rect
attr c  : color | black
```

図 1.2: figure.cm

このクラスモデルには 3 つのクラス fig, rect, crect が定義されている．fig は図形を表すクラスである．fig は 2 つの属性 x，y を持つ．それぞれ図形が位置する x 座標，y 座標を表す．両方とも整数型 int であり，デフォルト値は 0 である．デフォルト値とはオブジェクト生成直後に設定される値である．rect は fig の子クラスであり，長方形を表すクラスである．rect は 2 つの属性 w，h を持つ．それぞれ長方形の幅，高さを表す．両方とも int 型であり，デフォルト値は 0 である．crect は rect の子クラスであり，色つき長方形を表すクラスである．crect は色を表す属性 c を持つ．型は color であり，デフォルト値は black である．color 型はいくつかの色を要素に持つ列挙型である．

図 1.3 は，このクラスモデルを UML のクラス図で表したものである．

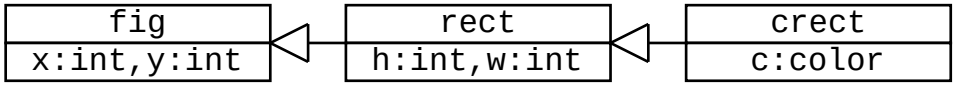

図 1.3: クラスモデルの例

1.2　ML におけるシミュレーション実行

モデルのシミュレーションを行う方法を説明する．シミュレーション実行を行うためには，objLib が提供する関数 mk_simulator により，クラスモデル figure.cm のシミュレー

タを生成する必要がある．

```
- mk_simulator("/.../figure.cm");
> val it = () : unit
```

これにより figure.cm が置かれるディレクトリに ML ストラクチャfigure.sml が生成される．このストラクチャの中にはオブジェクトを扱うための様々なプリミティブ演算子が含まれている．まず，プリミティブ演算子の動作について説明し，次にこれらを用いてメソッドを定義する方法を説明する．

1.2.1 プリミティブ演算子

シミュレータ figure.sml のシグネチャfigure.sig を図 1.4, 1.5 に示す．これらの型や関数を順に説明していく．

ストアとオブジェクト参照

型 store はシステムの環境を表す型である，直感的には，システム中に存在する全オブジェクトを格納するヒープメモリである．型 fig, rect, crect はオブジェクトの参照型である．定数 store_emp は空のストア，つまり，まだひとつもオブジェクトが生成されていないストアを表す．

次の 3 つの定数 fig_null, rect_null, crect_null は 3 つのクラスそれぞれの NULL オブジェクトの参照を表す．NULL オブジェクトは各クラスごとに異なる型を持っている．

次の 3 つの述語 fig_eq, rect_eq, crect_eq は，オブジェクト同士の同一性を検査する述語である．

new 演算子（オブジェクトの生成）

次の 3 つの関数 fig_new, rect_new, crect_new は，3 つのクラスそれぞれのオブジェクトを生成する関数である．これらを new 演算子と呼ぶ．例えば，rect_new は，引数のストアに対し rect オブジェクトを生成し，生成したオブジェクトと，生成後のストアのペアを返す．以下は rect_new の使用例である．

```
signature figure = sig
  type store
  type fig
  type rect
  type crect

  val store_emp : store

  val fig_null : fig
  val rect_null : rect
  val crect_null : crect

  val fig_eq : fig -> store -> bool
  val rect_eq : rect -> store -> bool
  val crect_eq : crect -> store -> bool

  val fig_new : store -> fig * store
  val rect_new : store -> rect * store
  val crect_new : store -> crect * store

  val fig_ex : fig -> store -> bool
  val rect_ex : rect -> store -> bool
  val crect_ex : crect -> store -> bool

  val fig_get_x : fig -> store -> int
  val fig_get_y : fig -> store -> int
  val rect_get_x : rect -> store -> int
  val rect_get_y : rect -> store -> int
  val rect_get_w : rect -> store -> int
  val rect_get_h : rect -> store -> int
  val crect_get_x : crect -> store -> int
  val crect_get_y : crect -> store -> int
  val crect_get_w : crect -> store -> int
  val crect_get_h : crect -> store -> int
  val crect_get_c : crect -> store -> color

  val fig_set_x : fig -> int -> store -> store
  val fig_set_y : fig -> int -> store -> store
  val rect_set_x : rect -> int -> store -> store
  val rect_set_y : rect -> int -> store -> store
  val rect_set_w : rect -> int -> store -> store
  val rect_set_h : rect -> int -> store -> store
  val crect_set_x : crect -> int -> store -> store
  val crect_set_y : crect -> int -> store -> store
  val crect_set_w : crect -> int -> store -> store
  val crect_set_h : crect -> int -> store -> store
  val crect_set_c : crect -> color -> store -> store

  ...
```

図 1.4: figure.sig (1)

```
  ...

  val fig_cast_rect : fig -> store -> rect
  val rect_cast_fig : rect -> store -> fig
  val fig_cast_crect : fig -> store -> crect
  val crect_cast_fig : crect -> store -> fig
  val rect_cast_crect : rect -> store -> crect
  val crect_cast_rect : crect -> store -> rect

  val fig_is_fig : fig -> store -> bool
  val fig_is_rect : fig -> store -> bool
  val fig_is_crect : fig -> store -> bool
  val rect_is_rect : rect -> store -> bool
  val rect_is_crect : rect -> store -> bool
  val crect_is_crect : crect -> store -> bool
end
```

図 1.5: figure.sig (2)

```
- val (r1,s) = rect_new store_emp;  (1)
> val r1 = <rect> : rect
> val s = <store> : store
- val (r2,s) = rect_new s;  (2)
> val r2 = <rect> : rect
> val s = <store> : store
- rect_eq r1 r2; (3)
> val it = false
- rect_eq r1 rect_null; (4)
> val it = false
```

(1) rect_new を空のストア store_emp に適用し，結果の値を変数 r1, s に格納する．r1 は生成された rect オブジェクトであり，s は生成後のストアである．オブジェクトやストアの内部構造は，不透明なシグネチャ制約により隠蔽されているため，その値は<rect>や<store>のように表示される．(2) さらにもう 1 つの rect オブジェクトを生成し r2 とする．生成後のストアを s とする．(3) rect_eq により，r1 と r2 が同一のオブジェクトであるかどうかを検査する．これらは別々に生成されたオブジェクトであるため，結果は false となる．(4) rect_eq により，r1 と rect_null が同一のオブジェクトであるかどうかを検査する．r1 は生成されストアに存在するオブジェクトであるため，結果は false となる．

ex 演算子（オブジェクトの存在検査）

述語 fig_ex, rect_ex, crect_ex は，3 つのクラスそれぞれのオブジェクトがストアに存在するかどうかを検査する述語である．これらを ex 演算子と呼ぶ．例えば，rect_ex は，第 1 引数の rect オブジェクトが第 2 引数のストアに存在するかどうかを検査する．以下は rect_ex の使用例である．

```
- rect_ex r1 s;  (1)
> val it = true : bool
- rect_ex r1 store_emp;  (2)
> val it = false : bool
- rect_ex rect_null s;  (3)
> val it = false : bool
```

(1) 新しく生成した rect オブジェクト r1 は，当然，生成後のストア s に存在するため，結果は true となる．(2) r1 は空のストア store_emp には存在しないため，結果は false となる．(3) NULL オブジェクトはいかなるストアにも存在しないしないため，結果は false となる．

get 演算子（属性の取得），set 演算子（属性の更新）

次の 11 個の関数は，オブジェクトの属性を取得する関数である．これらを get 演算子と呼ぶ．fig クラスについては，自身の属性 x, y に対応して 2 つの関数が定義される．例えば，fig_get_x は，第 2 引数のストアにおける第 1 引数の fig オブジェクトの属性 x の値を取得する関数である．rect クラスについては，継承する属性 x, y と自身の属性 w, h に対応して 4 つの関数が定義される．crect クラスについては，継承する属性 x, y, w, h と自身の属性 c に対応して 5 つの関数が定義される．

さらに次の 11 個の関数は，オブジェクトの属性を更新する関数である．これらの関数も各クラスの属性に対応して定義される．例えば，fig_set_x は，第 3 引数のストアにおいて，第 1 引数の fig オブジェクトの属性 x の値を第 2 引数の値に更新し，更新後のストアを返す関数である．

以下はこれらの関数の使用例である．

```
- rect_get_x r1 s;  (1)
> val it = 0 : int
- val s = rect_set_x r1 10 s (2)
> val s = <store> : store
- rect_get_x r1 s;  (3)
> val it = 10 : int
- rect_get_x rect_null s;  (4)
> !Uncaught exception:
!NotExists
```

(1) ストア s における r1 の属性 x の値はデフォルト値の 0 である．(2) r の属性 x を 10 に更新する．(3) 更新後のストア s において属性 x を取得すると結果は設定した値 10 となっている．(4) NULL オブジェクトの属性を取得した場合，例外 NotExists が発生する．NULL オブジェクトに限らず，ストアに存在しないオブジェクトの属性を取得した場合は必ずこの例外が発生する．

cast 演算子（オブジェクトの型変換）

次の 6 つの関数はオブジェクトの型変換（キャスト）を行う関数である．これらを cast 演算子と呼ぶ．cast 演算子により，オブジェクトの見かけの型を変更することが可能である．例えば，fig_cast_rect は fig 型から rect 型にダウンキャストする関数，rect_cast_fig は rect 型から fig 型にアップキャストする関数である．以下はこれらの関数の使用例である．

```
- val f = rect_cast_fig r1 s;  (1)
> val f = <fig> : fig
- val s = fig_get_x f s; (2)
> val it = 10 : int
- val r2 = fig_cast_rect f s;  (3)
> val r2 = <rect> : rect
- rect_eq r1 r2;  (4)
> val it = true : bool
- val c = rect_cast_crect r1 s;  (5)
> val c = <crect> : crect
- crect_eq c crect_null;  (6)
> val it = true
```

(1) rect_cast_fig により r1 を fig 型に変換し，f とする．(2) fig_get_x により f の属性 x を取得する．結果は 10 となる．これは先ほど f と同じオブジェクト r1 が設定した

値である．(3) fig_cast_rect により f を rect 型に変換し，r とする．(4) rect_eq により r1 と r2 の同一性を検査する．r2 は r1 をアップキャスト，ダウンキャストして得られるものであるから，結果は true となる．(5) rect_cast_crect により r1 を crect 型に変換し，c とする．r1 はもともと rect オブジェクトとして生成されたオブジェクトであるから，これは不正なダウンキャストである．(6) c は NULL オブジェクト crect_null と同一である．不正なダウンキャストによって得られるオブジェクトは変換先のクラスの NULL オブジェクトとなる．

is 演算子（インスタンス・オブ）

最後の 6 つの述語は，オブジェクトの実際の型（生成時の型）を判定する述語である．これらを is 演算子と呼ぶ．例えば，fig_is_fig, fig_is_rect, fig_is_crect は，fig 型のオブジェクトの実際の型がそれぞれ fig, rect, crect であるかどうかを検査する．以下はこれらの述語の使用例である．

```
- val (r,s) = rect_new s;  (1)
> val r = <rect> : rect
> val s = <store> : store
- rect_is_rect r s;  (2)
> val it = true : bool
- val f = rect_cast_fig r s; (3)
> val f = <fig> : fig
- fig_is_fig f s;  (4)
> val it = false : bool
- fig_is_rect f s;  (5)
> val it = true : bool
- fig_is_crect f s;  (6)
> val it = false : bool
```

(1) rect_new により rect オブジェクトを生成し，得られたオブジェクトとストアをそれぞれ r, s とする．(2) rect_is_rect により，r の実際の型が rect であるかどうかを検査する．r は rect_new によって生成されたので結果は true となる．(3) rect_cast_fig により r を fig 型にアップキャストし，f とする．(4)(5)(6) fig_is_fig, fig_is_rect, fig_is_crect により，f の実際の型がそれぞれ fig, rect, crect であるかどうかを検査する．結果は fig_is_rect の場合のみ true となる．このように，型変換によりオブジェクトの見かけの型が変わっても，これらの述語を用いて実際の型が一意に判別可能である．なお，is 演算子は，動的束縛をともなう仮想関数の定義に使用する（後述）．

1.2.2 メソッドの定義

以上のプリミティブ演算子を用いてメソッドを定義する方法を説明する．

fig クラスのメソッド

まず，fig クラスのオブジェクト生成メソッド new_fig を次のように定義する．

```
- fun fig_const f x y s = fig_set_y f y (fig_set_x f x s);
> val fig_const = fn : fig -> int -> int -> store -> store
- fun new_fig x y s =
    let
      val (f,s) = fig_new s
      val s = fig_const f x y s
    in
      (f,s)
    end;
> val new_fig = fn : int -> int -> store -> fig * store
```

fig_const はコンストラクタであり，fig オブジェクト f と，int 型の値 x, y を入力とし，fig_set_x, fig_set_y を用いてそれぞれ f の属性 x, y の値に設定する．new_fig は，fig_new を用いて fig オブジェクトを生成し，コンストラクタを適用する．出力は，生成した fig オブジェクトと，生成後のストアである．

次に，座標を取得するメソッド fig_getPos を次のように定義する[1]．

```
- fun fig_getPos f s = (fig_get_x f s, fig_get_y f s);
> val fig_getPos = fn : fig -> store -> int * int
```

fig_getPos は，fig_get_x, fig_get_y を用いて属性 x, y の値を取得し，それらをペアにして返す．

次に，図形を移動させるメソッド fig_move を次のように定義する．

[1] クラス c のメソッド m の名前は慣習的に，c_m とする．また，オブジェクト生成メソッドは new_c，コンストラクタは c_const とする

```
- fun fig_move f dx dy s =
    let
      val x = fig_get_x f s
      val y = fig_get_y f s
      val s = fig_set_x f (x+dx) s
    in
      fig_set_y f (y+dy) s
    end;
> val fig_move = fn : fig -> int -> int -> store -> store
```

fig_move は，fig_get_x, fig_get_y により属性 x, y の値を取得し，それぞれに dx, dy を加えた値を fig_set_x, fig_set_y により設定する．

これらのメソッドの使用例を以下に示す．

```
- val (f,s) = new_fig 5 3 store_emp;
> val f = <fig> : fig
> val s = <store> : store
- fig_getPos f s;
> val it = (5,3) : int * int
- val s = fig_move f (~2) 3 s;
> val s = <store> : store
- fig_getPos f s;
> val it = (3,6) : int * int
```

さらに x 軸対称移動，y 軸対称移動，原点対称移動を行うメソッド fig_symX, fig_symY, fig_sym0 を次のように定義する．

```
- fun fig_symX f s = fig_set_y f (~(fig_get_y f s)) s;
> val fig_symX = fn : fig -> store -> store
- fun fig_symY f s = fig_set_x f (~(fig_get_x f s)) s;
> val fig_symY = fn : fig -> store -> store
- fun fig_sym0 f s = fig_symY f (fig_symX f s);
> val fig_sym0 = fn : fig -> store -> store
```

fig_symX, fig_symY は，それぞれ属性 y, x の値の正負を反転させる．fig_sym0 は，それら 2 つを連続して行う．これらのメソッドの使用例を以下に示す．

```
- val s = fig_symX f s;
> val s = <store> : store
- fig_getPos f s;
> val it = (3,~6) : int * int
- val s = fig_symY f s;
> val s = <store> : store
- fig_getPos f s;
> val it = (~3,~6) : int * int
- val s = fig_symO f s;
> val s = <store> : store
- fig_getPos f s;
> val it = (3,6) : int * int
```

rect クラスのメソッド

次に rect クラスのメソッドを定義する．rect クラスのメソッド定義においては fig クラスのメソッドを継承または，オーバーライドすることにより行う．この体系では継承，オーバーライドを実現するための明示的な文法は存在しないが，子クラスから親クラスのメソッドを呼び出すことは，型変換を利用して実現することが可能である．

オブジェクト生成メソッド new_rect は次のように定義する．

```
- fun rect_const r x y w h s =
    let
      val s = fig_const (rect_cast_fig r s) x y s
    in
      rect_set_h r h (rect_set_w r w s)
    end;
> val rect_const = fn :
    rect -> int -> int -> int -> int -> store -> store
- fun new_rect x y w h s =
    let
      val (r,s) = rect_new s
      val s = rect_const r x y w h s
    in
      (r,s)
    end;
> val new_rect = fn :
    int -> int -> int -> int -> store -> rect * store
```

コンストラクタ rect_const は，4 つの属性 x, y, w, h の値を設定する．属性 x, y の値の設定は親クラスのコンストラクタを呼び出すことにより行う．親クラスのメソッドを呼び出すためには，rect_cast_fig により rect オブジェクト r を fig オブジェクトに変換し，fig_const を適用する．属性 w, h の値の設定は，rect_set_w, rect_set_h により行う．

座標を取得するメソッド rect_getPos，図形を移動するメソッド rect_move はそれぞれ fig クラスの対応するメソッド fig_getPos, fig_move を継承することにより定義する．これは次のように行う．

```
- fun rect_getPos r s = fig_getPos (rect_cast_fig r s) s;
> val rect_getPos = fn : rect -> store -> int * int
- fun rect_move r dx dy s = fig_move (rect_cast_fig r s) dx dy s;
> val rect_move = fn : rect -> int -> int -> store -> store
```

rect_getPos は，rect オブジェクト r を rect_cast_fig を用いて fig オブジェクトに変換し，fig_getPos を適用する．rect_move についても同様である．このようにメソッドの継承は親クラスの型に変換し，親クラスのメソッドを呼び出すことにより実現することが可能である．

3 つの対称移動メソッド rect_symX, rect_symY, rect_symO についてはオーバーライドを行う．オーバーライドは，メソッドの中身を新しく定義するだけである．その際に，親クラスのメソッドを呼び出す場合は，メソッドの継承と同様に型変換を行う．

```
- fun rect_symX r s =
    let
      val s = fig_symX (rect_cast_fig r s) s
    in
      rect_move r 0 (~(rect_get_h r s)) s
    end;
> val rect_symX = fn : rect -> store -> store
- fun rect_symY r s =
    let
      val s = fig_symY (rect_cast_fig r s) s
    in
      rect_move r (~(rect_get_w r s)) 0 s
    end;
> val rect_symY = fn : rect -> store -> store
- fun rect_sym0 r s = rect_symY r (rect_symX r s);
> val rect_sym0 = fn : rect -> store -> store
```

rect_symX は，まず，親クラスのメソッド fig_symX を用いて，基準点（長方形の左下の点）の x 軸対称移動を行い，次に，長方形の高さ分だけ y 軸の負の方向に移動する．rect_symX は，まず，基準点の y 軸対称移動を行い，次に，長方形の幅分だけ x 軸の負の方向に移動する．rect_sym0 はこれらを連続して適用する（図 1.6）．これらのメソッドの使用例を以下に示す．

```
- val (r,s) = new_rect 3 2 6 3 s
> val r = <rect> : rect
> val s = <store> : store
- val s = rect_symX r s;
> val s = <store> : store
- rect_getPos r s;
> val it = (3,~5) : int * int
- val s = rect_symY r s;
> val s = <store> : store
- rect_getPos r s;
> val it = (~9,~5) : int * int
- val s = rect_sym0 f s;
> val s = <store> : store
- rect_getPos r s;
> val it = (3,2) : int * int
```

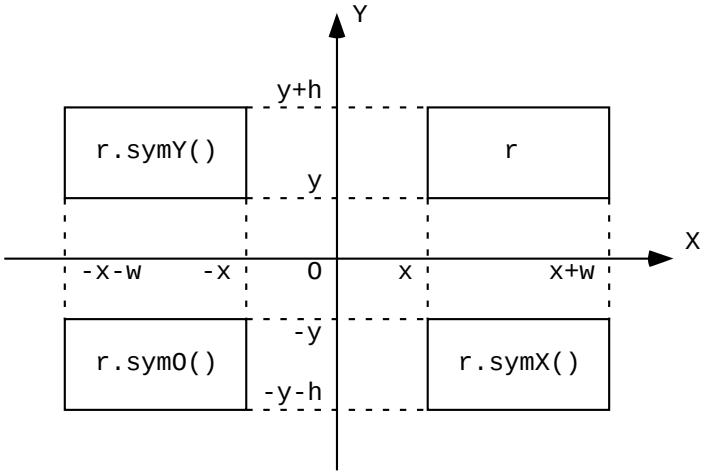

図 1.6: rect クラスのメソッド symX(), symY(), symO()

crect クラスのメソッド

crect クラスのオブジェクト生成メソッド new_crect は，new_rect と同様の方法で定義する．また，その他のメソッド crect_getPos, crect_move, crect_symX, crect_symY, crect_symO は，rect クラスの対応するメソッドを継承して定義する．

仮想関数と動的束縛

次に仮想関数を定義する．仮想関数とは，いわゆる generic function であり，適用されるオブジェクトがどのクラスのインスタンスであるかに応じて，その関数本体を切り替える関数である．このメカニズムは，is 演算子を用いて実現可能である．

v_fig_symO を fig オブジェクトに対する原点対称移動メソッドの仮想関数版であるとする．これは次のように定義される．

```
- fun v_fig_symO f s =
    if fig_is_fig f s then fig_symO f s
    else if fig_is_rect f s then rect_symO (fig_cast_rect f s) s
    else if fig_is_crect f s then crect_symO (fig_cast_crect f s) s
    else s
> val v_fig_symO = fn : fig -> store -> store
```

この関数は，適用される fig オブジェクト f が 3 つのサブクラス fig, rect, crect のどのクラスのインスタンスであるかを fig_is_fig, fig_is_rect, fig_is_crect により判定し，対応するメソッドを呼び出す．図 1.7 はこれを図示したものである．

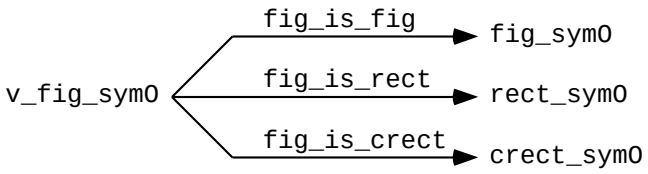

図 1.7: 動的束縛の実現

この関数の動作を以下に示す．

```
- val (f1,s) = new_fig 3 2 s  (1)
> val f1 = <fig> : fig
> val s = <store> : store
- val (r,s) = new_rect 3 2 6 3 s  (2)
> val r = <rect> : rect
> val s = <store> : store
- val f2 = rect_cast_fig r s;  (3)
> val f2 = <fig> : fig
- val s = v_fig_sym0 f1 s;  (4)
> val s = <store> : store
- fig_getPos f1 s;  (5)
> val it = (~3,~2) : int * int
- val s = v_fig_sym0 f2 s;  (6)
> val s = <store> : store
- fig_getPos f2 s;  (7)
> val it = (~9,~5) : int * int
```

(1) 座標位置 (3,2) の fig オブジェクトを生成し，f1 とする．(2) 座標位置 (3,2)，幅 6, 高さ 3 の rect オブジェクトを生成し，r とする．(3) r を fig 型に変換し f2 とする．(4) f1 の実際の型は fig であるから，ここでの v_fig_sym0 の実体は fig_sym0 である．(5) したがって，座標位置は (~3,~2) となる．(6) f2 の実際の型は rect であるから，ここでの v_fig_sym0 の実体は rect_sym0 である．(5) したがって，座標位置は (~9,~5) となる．

継承やオーバーライド，動的束縛などの基本的なオブジェクト指向概念は以上のように実現することができる．しかし，この体系ではコードの再利用性，拡張性，保守性といったオブジェクト指向が本来もたらすべき利便性が実現されていない．例えば，Java では，メソッドを継承する際は，新たにコードを書き加えなくてよいが，ObjectLogic の体系では，親クラスのメソッドを呼び出すことを明示的に定義しなければならない．また，Java では動的束縛を自動的に行ってくれるが，この体系では，if 文でスイッチを行う仮想関数を書かなくてはならない．さらにサブクラスが追加されるごとにその仮想関数を書き直さ

なければならない．Java の public, private といったカプセル化の概念もこの体系には存在しない．その他，抽象メソッド，インターフェースなどの概念も存在しない．

しかし，この体系がこのような利便性を備えていなくても，本来の目的である定理証明を行う上では一切問題ない．この体系は，実際のシステムを開発するためのプログラミング言語ではなく，システムの振る舞いを定理証明により検証することを目的としている．したがって，コードの再利用性や拡張性に優れていなくても，継承や，動的束縛の動作をシミュレートすることさえできればシステムの振る舞いの検証を行うという本来の目的のためには十分である．

それでも，シミュレータでメソッド定義をしていく際はそのような利便性があったほうがよい．この場合は，シミュレータの言語へと変換できるような何らかのオブジェクト指向言語を定義し，その言語レベルで利便性を実現すればよい．

1.3 HOL における定理証明

モデルの定理証明を行う方法を説明する．定理証明を行うためには，`objLib` が提供する関数 `mk_theory_script` により，クラスモデル `figure.cm` に対応する HOL 理論を生成する必要がある．

```
- mk_theory_script("/.../figure.cm");
> val it = () : unit
```

これにより，`figure.cm` が置かれるディレクトリに `figureScript.sml` という理論スクリプトファイルが生成される．理論スクリプトファイルとは，HOL において理論を生成するために必要な一連の HOL スクリプトが記述された ML ストラクチャである．これが置かれたディレクトリにおいて，HOL が提供する `Holmake` というシェルコマンドを実行すれば，HOL 理論 `figureTheory.sml` が生成される．この理論には，オブジェクト指向概念を表す様々な型，定数が含まれている．公理は `objLib` を通して取得することができる．まず，理論に含まれる型，演算子，公理について説明し，次に，メソッドを定義し，定理証明を行う方法を説明する．

1.3.1 型，演算子，公理

HOL 理論 `figureTheory.sml` をロードすれば，その中に定義される型や演算子が使用できるようになる．

```
- load "figureTheory.sml";
> val it = () : unit
```

`figureTheory.sml` には，シミュレータ `figure.sml` に含まれる型や演算子がそのままの名前で導入されている．例えば，`store` や `fig` などの型や，`fig_new`, `fig_get_x` などの演

算子が存在する[2]．それらを HOL において扱うためには``:store``や``fig_get_x``のように``...``で囲む必要がある．HOL における型，演算子は，ML においてそれぞれ hol_type, term という型を持つ．以下は，``:store``，``fig_get_x``を評価した様子である．

```
- ``:store``;
> val it = ``:store`` : hol_type
- ``:fig``;
> val it = ``:fig`` : hol_type
- ``fig_new``;
> val it = ``fig_new`` : term
- ``fig_get_x``;
> val it = ``fig_get_x`` : term
```

演算子の HOL における型を知りたいときは ML 関数 type_of を用いる．

```
- type_of ``fig_new``;
> val it = ``:store -> fig # store`` : hol_type
- type_of ``fig_get_x``;
> val it = ``:fig -> store -> int`` : hol_type
```

これらの HOL レベルの演算子は，シミュレータで行ったように実行することはできない．HOL では，これらの演算子の動作は「公理」として特徴付けられている．例えば，ストアにある fig オブジェクトが存在するとする．このオブジェクトに対し，fig_set_x により属性 x の値を更新し，さらにその直後に fig_get_x を適用すれば，その更新した値が返ってくる．この性質は次の公理で表される．

```
- objLib.GET_SET{Get=``fig_get_x``};
> val it =
  |- !i x s. fig_ex i s ==>
       (fig_get_x i (fig_set_x i x s) = x) : thm
```

この公理は,「fig オブジェクト i がストア s に存在するならば，s において i の属性 x をある値 x に更新し，更新後のストアにおいて同一の属性の値を取得する場合，その値は x となる」を意味する．GET_SET は，引数で与えられる get 演算子に関する上の性質を表す公理を取得する関数である．thm は定理を表す型である（公理や定義も含む）．

rect_get_w に関しても同様の性質の公理を取得したいときは次のようにすればよい．

[2]同一性検査 fig_eq, rect_eq, crect_eq は含まれていない．HOL では「=」を使う．

```
- objLib.GET_SET{Get=``rect_get_w``};
> val it =
  |- !i x s. rect_ex i s ==>
       (rect_get_w i (rect_set_w i x s) = x) : thm
```

このような公理は約 40 種類存在する．そのうちのいくつかを以下に示す．

次の 2 つの公理は，GET_SET と同様，属性の取得と更新に関する公理である．

```
- objLib.DIFF_OBJ_GET_SET{Get=``fig_get_x``};  (1)
> val it =
  |- !i j x s. ~(i = j) ==>
       (fig_get_x i (fig_set_x j x s) = fig_get_x i s) : thm
- objLib.DIFF_GET_SET{Get=``fig_get_x``,Set=``fig_set_y``};  (2)
> val it =
  |- !i j x s. fig_get_x i (fig_set_y j x s) = fig_get_x i s : thm
```

(1)2 つの fig オブジェクト i, j が異なるならば，j の属性 x の更新後に i の属性 x を取得して得られる値は，更新前に得られる値に等しい．つまり，あるオブジェクトの属性取得は，それとは異なるオブジェクトの属性更新に無関係である．(2) fig オブジェクト i の属性 y の更新後に j の属性 x を取得して得られる値は，更新前に得られる値に等しい．つまり，ある属性の取得は，それとは異なる属性の更新に無関係である．

次の 2 つの公理は cast 演算子に関するものである．

```
- objLib.UP_DOWN{Up=``rect_cast_fig``};  (1)
> val it =
  |- !i x s. rect_ex i s ==>
       (fig_cast_rect i (rect_cast_fig i s) = i) : thm
- objLib.UP_11{Up=``rect_cast_fig``};  (2)
> val it =
  |- !i x s. rect_ex i s /\ rect_ex j s ==>
       ~(i = j) ==>
       ~(rect_cast_fig i s = rect_cast_fig j s) : thm
```

(1) rect オブジェクトがストアに存在するとき，それを fig クラスにアップキャストし，さらに rect クラスにダウンキャストすれば元のオブジェクトと同一となる．つまり，型変換は可逆的である（図 1.8 左）．(2) 2 つの rect オブジェクトがストアに存在し，それらが異なるオブジェクトならば，それらを fig クラスにアップキャストして得られる 2 つの fig オブジェクトも異なるオブジェクトである．つまり，型変換は 1 対 1 関数である（図 1.9 右）．

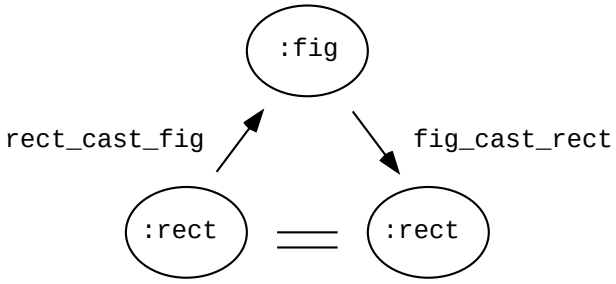

図 1.8: UP_DOWN

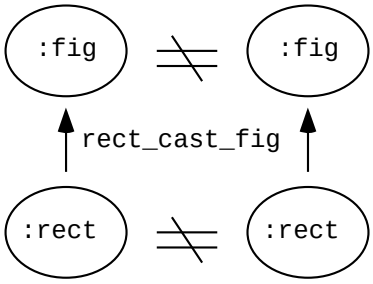

図 1.9: UP_11

次の 2 つの公理は，is 演算子に関するものである．

```
- objLib.IS_CAST{Is1=``rect_is_rect``,Is2=``fig_is_rect``};  (1)
> val it =
  |- !i s. rect_is_rect i s ==>
       fig_is_rect (rect_cast_fig i s) s : thm
- objLib.IS_IMP_NOT_IS
  {Is1=``rect_is_rect``,Is2=``rect_is_crect``};  (2)
> val it = |- !i s. rect_is_rect i s ==> ~rect_is_crect i s : thm
```

(1) rect オブジェクトの実際の型が rect であれば，それを親クラス fig に変換しても，実際の型は rect である．つまり，見かけの型が変化しても生成時の型は不変である（図 ??）．(2) rect オブジェクトが rect クラスのインスタンスならば，crect クラスのインスタンスではない．つまり，実際の型は唯一である．

1.3.2 メソッドの定義

シミュレータで定義したのと同様，HOL においてもプリミティブ演算子を用いてメソッドを定義する．HOL における関数の定義は次のように行う．

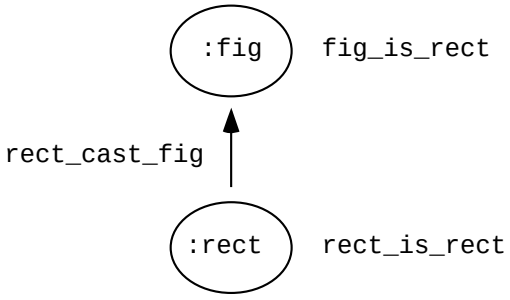

図 1.10: IS_CAST

```
- val fig_getPos = Define
    `fig_getPos f s = (fig_get_x f s, fig_get_y f s)`;
> val fig_getPos =
    |- fig_getPos f s = (fig_get_x f s, fig_get_y f s) : thm
- type_of ``fig_getPos``;
> val it = ``:fig -> store -> int # int`` : hol_type
```

Define は引数の`...`で囲まれた等式を新しい定理として返す．その他のメソッド 定義を図 1.11 に示す．

1.3.3 証明例

いくつかの証明例を以下に示す．

証明例 (1)

以下は fig オブジェクトに対する fig_move 適用後の x 座標の値の増加に関する命題である．

```
!f x y s. fig_ex f s ==>
  (fig_get_x f (fig_move f x y s) = fig_get_x f s + x)
```

これを Goal-Oriented Proof により証明する．Goal-Oriented Proof とは，証明対象の命題をゴールに設定し，タクティックと呼ばれる証明戦略により，より簡単なゴールへと分解していくという証明法である．

ゴールの設定は ML 関数 g により行う．g は，`...`で囲まれた命題をゴールに設定する．

```
val fig_move = Define
  `fig_move f dx dy s =
     let x = fig_get_x f s in
     let y = fig_get_y f s in
     let s = fig_set_x f (x+dx) s in
       fig_set_y f (y+dy) s`;

val fig_symX = Define `fig_symX f s = fig_set_y f (~(fig_get_y f s))`;

val fig_symY = Define `fig_symY f s = fig_set_x f (~(fig_get_x f s)) s`;

val fig_symO = Define `fig_symO f s = fig_symY f (fig_symX f s)`;

val rect_getPos = Define `rect_getPos r s = fig_getPos (rect_cast_fig r s) s`;

val rect_move = Define
  `rect_move r dx dy s = fig_move (rect_cast_fig r s) dx dy s`;

val rect_symX = Define
  `rect_symX r s =
     rect_move r (~(rect_get_h r s)) (fig_symX (rect_cast_fig r s) s)`;

val rect_symY = Define
  `rect_symY r s =
     rect_move r (~(rect_get_w r s)) (fig_symY (rect_cast_fig r s) s)`;

val rect_symO = Define `rect_symO r s = rect_symY r (rect_symX r s)`;

val crect_getPos = Define
  `crect_getPos r s = rect_getPos (crect_cast_rect c s) s`;

val crect_move = Define
  `crect_move r dx dy s = rect_move (crect_cast_rect c s) dx dy s`;

val crect_symX = Define `crect_symX c s = rect_symX (crect_cast_rect c s) s`;

val crect_symY = Define `crect_symY c s = rect_symY (crect_cast_rect c s) s`;

val crect_symO = Define `crect_symO c s = rect_symO (crect_cast_rect c s) s`;

val v_fig_symO f s = Define
  `v_fig_symO f s =
     if fig_is_fig f s then fig_symO f s
     else if fig_is_rect f s then rect_symO (fig_cast_rect f s) s
     else if fig_is_crect f s then crect_symO (fig_cast_crect f s) s
     else s`;
```

図 1.11: HOL **におけるメソッドの定義**

```
- g `!f x y s. fig_ex f s ==>
    (fig_get_x f (fig_move f x y s) = fig_get_x f s + x)`;
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
       Initial goal:
       !f x y s. fig_ex f s ==>
         (fig_get_x f (fig_move f x y s) = fig_get_x f s + x)
   : proofs
```

ゴールが設定されると，タクティックが適用できるようになる．タクティックの適用は ML 関数 e により行う．まず，fig_move の定義により書き換えを行う．

```
- e (REWRITE_TAC [fig_move]);
OK..
1 subgoal:
> val it =
    !f x y s. fig_ex f s ==>
      (fig_get_x f
        (let x' = fig_get_x f s in
           let y' = fig_get_y f s in
             let s = fig_set_x f (x' + x) s in
               fig_set_y f (y' + y) s) =
       fig_get_x f s + x)
   : goalstack
```

REWRITE_TAC は引数のリストに含まれる定理群によりゴールの書き換えを行う．

次に let の項を展開する．

```
- e LET_TAC;
OK..
1 subgoal:
> val it =
    !f x y s. fig_ex f s ==>
      (fig_get_x f
        (fig_set_y f (fig_get_y f s + y)
          (fig_set_x f (fig_get_x f s + x) s)) =
       fig_get_x f s + x)
   : goalstack
```

LET_TAC は次のように定義されるタクティックであり，let a = b in ... の形の項を展開する．

```
val LET_TAC = CONV_TAC (DEPTH_CONV let_CONV);
```

ここで，ストアに対して fig_set_y と fig_get_x が連続して適用されている部分に着目する．属性 x と y は異なる属性であるから，fig_get_x によって取得される値は fig_set_y による更新に無関係である．したがって，DIFF_GET_SET により書き換えを行う．

```
- e (REWRITE_TAC
     [objLib.DIFF_GET_SET{Get=``fig_get_x``,Set=``fig_set_y``}]);
OK..
1 subgoal:
> val it =
    !f x s. fig_ex f s ==>
      (fig_get_x f (fig_set_x f (fig_get_x f s + x) s) =
       fig_get_x f s + x)
   : goalstack
```

これは公理 GET_SET{Get=``fig_get_x``}そのものであるから，この公理によって書き換えを行うことにより証明を完了することができる．

```
- e (REWRITE_TAC [objLib.GET_SET{Get=``fig_get_x``}]);
OK..
> val it =
  Initial goal proved.
  |- !f x y s. fig_ex f s ==>
       (fig_get_x f (fig_move f x y s) = fig_get_x f s + x)
   : goalstack
```

証明した定理は次のように獲得することができる．

```
- val fig_get_x_fig_move = top_thm();
> val fig_get_x_fig_move =
    |- !f x y s. fig_ex f s ==>
         (fig_get_x f (fig_move f x y s) = fig_get_x f s + x) : thm
```

証明とその状態（完了，未完了）はスタックにより管理されており，現在行っている証明がスタックの先頭に位置している．ML 関数 `top_thm:unit->thm` は，その先頭の定理を取得する関数である．

証明例 (2)

以下は `rect` オブジェクトに対する `rect_move` 適用後の `x` 座標の値の増加に関する命題である．

```
!r x y s. rect_ex r s ==>
  (rect_get_x r (rect_move r x y s) = rect_get_x r s + x)
```

`rect_move` は親クラスのメソッド `fig_move` を用いて定義されているため，(1) で証明した `fig_move` に関する定理を補題として利用することができる（証明の継承）．

まず，補題として次の命題を証明する．

```
!f r x y s. rect_cast_fig r (fig_move f x y s) = rect_cast_fig r s
```

これは，`rect` オブジェクトの型変換が `fig_move` の適用に無関係であることを表す命題である．

```
- g `!f r x y s.
       rect_cast_fig r (fig_move f x y s) = rect_cast_fig r s`:
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
       Initial goal:
       !f r x y s.
         rect_cast_fig r (fig_move f x y s) = rect_cast_fig r s
   : proofs
```

まず，`fig_move` の定義の展開と let の展開を行う．

```
- e (REWRITE_TAC [fig_move] THEN LET_TAC);
OK..
1 subgoal:
> val it =
    !f r x y s.
      rect_cast_fig r
        (fig_set_y f (fig_get_y f s + y)
         (fig_set_x f (fig_get_x f s + x) s)) =
      rect_cast_fig r s
   : goalstack
```

THEN は，t1 THEN t2 のように 2 つのタクティックを引数とし，それらを続けて行うタクティックを構成する．このような，タクティックを引数にとり新しいタクティックを構成する関数はタクティカルと呼ばれる．

ここで，fig_set_y の適用後に rect_cast_fig が適用されている部分に着目する．属性の更新とオブジェクトの型変換は無関係であるから，fig_set_y の適用は省略することができる．この性質は次の公理によって表される．

```
- objLib.CAST_SET{Cast=``rect_cast_fig``,Set=``fig_set_y``};
> val it =
  |- !i j x s.
     rect_cast_fig i (fig_set_y j x s) = rect_cast_fig i s : thm
```

この公理により書き換えを行う．

```
- e (REWRITE_TAC
     [objLib.CAST_SET{Cast=``rect_cast_fig``,Set=``fig_set_y``}]);
OK..
1 subgoal:
> val it =
    !f r x y s.
      rect_cast_fig r (fig_set_x f (fig_get_x f s + x) s)) =
      rect_cast_fig r s
   : goalstack
```

fig_set_y と rect_cast_fig についても同様の公理について書き換えを行うことにより，証明を完了することができる．

```
- e (REWRITE_TAC
     [objLib.CAST_SET{Cast=``rect_cast_fig``,Set=``fig_set_x``}]);
OK..
> val it =
  Initial goal proved.
  |- !f r x y s.
     rect_cast_fig r (fig_move f x y s) = rect_cast_fig r s
   : goalstack
```

証明された定理を rect_cast_fig_fig_move とする．

```
- val rect_cast_fig_fig_move = top_thm();
> val rect_cast_fig_fig_move =
    |- !f r x y s.
         rect_cast_fig r (fig_move f x y s) = rect_cast_fig r s : thm
```

次に，目的の定理の証明を行う．

```
- g `!r x y s. rect_ex r s ==>
     (rect_get_x r (rect_move r x y s) = rect_get_x r s + x)`;
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
       Initial goal:
       !r x y s. rect_ex r s ==>
         (rect_get_x r (rect_move r x y s) = rect_get_x r s + x)
   : proofs
```

まず，全称量子の除去，前提条件の仮定リストへの移動を行う．

```
- e (REPEAT STRIP_TAC);
OK..
1 subgoal:
> val it =
    rect_get_x r (rect_move r x y s) = rect_get_x r s + x
    ------------------------------------
      rect_ex r s
   : goalstack
```

ここで，-----の上側がゴールの結論，下側が仮定のリストである．STRIP_TAC は，全称

量子の除去や前提条件の仮定リストへの移動を一度だけ行うタクティックである．REPEAT は引数で与えられるタクティックを適用可能な限り繰り返すタクティカルである．したがって，REPEAT STRIP_TAC により，4 つの全称量子がすべて除去され，さらに前提条件が仮定リストに移動される．

次に，rect_move の定義により書き換えを行う．

```
- e (REWRITE_TAC [rect_move] THEN LET_TAC);
OK..
1 subgoal:
> val it =
    rect_get_x r (fig_move (rect_cast_fig r s) x y s) =
    rect_get_x r s + x
    ------------------------------------
      rect_ex r s
   : goalstack
```

ここで，rect_get_x について次の公理が存在する．

```
- objLib.SPR_GET{Get=``rect_get_x``};
> val it =
  |- !i s. rect_get_x i s = fig_get_x (rect_cast_fig i s) s : thm
```

この公理は，rect オブジェクトの属性 x の値を取得することは，その rect オブジェクトを fig オブジェクトに変換してから取得することと等しいことを意味する．

この公理により書き換えを行う．

```
- e (REWRITE_TAC [objLib.SPR_GET{Get=``rect_get_x``}]);
OK..
1 subgoal:
> val it =
    fig_get_x (rect_cast_fig r (fig_move (rect_cast_fig r s) x y s))
      (fig_move (rect_cast_fig r s) x y s) =
    fig_get_x (rect_cast_fig r s) s + x
    ------------------------------------
      rect_ex r s
   : goalstack
```

ここで，前に証明しておいた補題 rect_cast_fig_fig_move により書き換えを行う．

```
- e (REWRITE_TAC [rect_cast_fig_fig_move]);
OK..
1 subgoal:
> val it =
    fig_get_x (rect_cast_fig r s)
      (fig_move (rect_cast_fig r s) x y s) =
    fig_get_x (rect_cast_fig r s) s + x
    ------------------------------------
      rect_ex r s
   : goalstack
```

ここで，rect_ex について次の定理が存在する．

```
- objLib.EX_IMP_SPR_EX{Up=``rect_cast_fig``};
> val it =
  |- !i s. rect_ex i s ==> fig_ex (rect_cast_fig i s) s : thm
```

この定理は，rect オブジェクトがストアに存在するならば，それを親クラスである fig クラスに変換したオブジェクトもストアに存在することを意味する．この定理の結論部をゴールの仮定から導出する．

```
- e (IMP_RES_TAC (objLib.EX_IMP_SPR_EX{Up=``rect_cast_fig``}));
OK..
1 subgoal:
> val it =
    fig_get_x (rect_cast_fig r s)
      (fig_move (rect_cast_fig r s) x y s) =
    fig_get_x (rect_cast_fig r s) s + x
    ------------------------------------
      0. rect_ex r s
      1. fig_ex (rect_cast_fig r s) s
   : goalstack
```

IMP_RES_TAC は，引数で与えられる定理とゴールの仮定に対し，モーダス・ポーネンズを適用し，得られる結論をゴールの仮定に追加するタクティックである．

最後に，(1) で証明した定理 fig_get_x_fig_move を適用し，証明を完了する．

```
- e (ASM_SIMP_TAC bool_ss [fig_get_x_fig_move]);
OK..
> val it =
    Initial goal proved.
    |- !r x y s. rect_ex r s ==>
         (rect_get_x r (rect_move r x y s) = rect_get_x r s + x)
      : goalstack
```

ASM_SIMP_TAC:simpset->thm list->tactic は，第 1 引数の simpset(simplifier set) と，第 2 引数の定理のリスト，さらにゴールの仮定のリストにより，簡単化を行う．simpset とは，簡約に用いるための定理を集めたものであり，例えば，bool_ss にはブール代数に関する定理群が含まれている．他に，リストの定理集合 list_ss や算術理論の定理集合 arith_ss などがある．

高級タクティックによる証明

証明例 (1)(2) では，GET_SET や DIFF_GET_SET などの公理により直接書き換えを行っていた．しかし，公理の数は非常に多く，ユーザがそれらをすべて把握し，1 つづつ細かく適用していくのは効率が悪い．そこで，objLib ではゴールの形に応じて自動的に必要な公理を選定し，適用するタクティックを用意している．

主なタクティックは，SLICE_TAC, OBJ_SIMP_TAC の 2 つである．SLICE_TAC はスライサの役目を持つタクティックであり，ex 演算子や get 演算子の値に影響しない new 演算子や set 演算子を除去する機能を持つ．例えば，ゴールに fig_get_x f (fig_set_y f y s) という項が含まれているならば，公理 DIFF_GET_SET を自動的に適用し，fig_get_x f s に簡略化する．OBJ_SIMP_TAC は，スライサの適用によって残される本質的な部分の証明を試みるタクティックである．例えば，ゴールに fig_get_x f (fig_set_x f x s) という項が含まれているならば，公理 GET_SET の適用を試みる．これら 2 つのタクティックを用いれば，証明の流れは基本的に，「定義の展開→ SLICE_TAC の適用→ OBJ_SIMP_TAC の適用」だけで済む．

これらのタクティックを用いてもう一度証明例 (2) の定理を証明する．

```
- g `!r x y s. rect_ex r s ==>
     (rect_get_x r (rect_move r x y s) = rect_get_x r s + x)`;
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
       Initial goal:
       !r x y s. rect_ex r s ==>
         (rect_get_x r (rect_move r x y s) = rect_get_x r s + x)
   : proofs
```

まず，定義をすべて展開する．

```
- e (REWRITE_TAC [rect_move, fig_move] THEN LET_TAC);
> val it =
  !r x y s.
    rect_ex r s ==>
      (rect_get_x r   (1)
        (fig_set_y (rect_cast_fig r s)
          (fig_get_y (rect_cast_fig r s) s + y)
          (fig_set_x (rect_cast_fig r s)
            (fig_get_x (rect_cast_fig r s) s + x) s)) =
       rect_get_x r s + x)  (2)
   : goalstack
```

ここで，(1)(2) における get 演算子 rect_get_x に着目する．これは公理 SPR_GET が適用できる項である．このように，祖先クラスで定義される属性を取得，更新する get 演算子，set 演算子が存在するとき，SPR_TAC を適用すればよい．SPR_TAC はゴールに含まれるそのような項をすべて書き換える機能を持つ．

```
- e SPR_TAC;
> val it =
  !r x y s.
    rect_ex r s ==>
      (fig_get_x  (1)
        (rect_cast_fig r  (2)
          (fig_set_y (rect_cast_fig r s)  (3)
            (fig_get_y (rect_cast_fig r s) s + y)
            (fig_set_x (rect_cast_fig r s)  (4)
              (fig_get_x (rect_cast_fig r s) s + x) s)))
        (fig_set_y (rect_cast_fig r s)  (5)
          (fig_get_y (rect_cast_fig r s) s + y)
          (fig_set_x (rect_cast_fig r s)
            (fig_get_x (rect_cast_fig r s) s + x) s)) =
       fig_get_x (rect_cast_fig r s) s + x)
   : goalstack
```

ここで，(1)(5) について，get 演算子 fig_get_x と set 演算子 fig_set_y は異なる属性に対する取得と更新であるから，公理 DIFF_GET_SET により簡単化することができる．また，(2)(3)(4) について，cast 演算子 rect_cast_fig は，set 演算子 fig_set_y, fig_set_x の適用に無関係であるから，公理 CAST_SET により簡単化することができる．このように，get 演算子や cast 演算子の値に影響しない set 演算子や new 演算子が存在するとき，SLICE_TAC を適用すればよい．SLICE_TAC はこのような証明の本質に無関係な演算子を一掃することができる．

```
- e SLICE_TAC;
> val it =
  !r x y s.
    rect_ex r s ==>  (1)
      (fig_get_x (rect_cast_fig r s)  (2)
        (fig_set_x (rect_cast_fig r s)  (3)
          (fig_get_x (rect_cast_fig r s) s + x) s)) =
       fig_get_x (rect_cast_fig r s) s + x)
   : goalstack
```

スライサにより残されたゴールは証明の本質的な部分である．これを証明するためには，まず，(1) の rect_ex の項から定理 EX_IMP_SPR_EX により，オブジェクト rect_cast_fig r s がストアに存在するという仮定を導出し，次に，この仮定と公理 GET_SET により．(2)(3) の同一の属性についての get 演算子，set 演算子を簡単化すればよい．このような証明は

OBJ_SIMP_TAC が自動的に行う．

```
- e OBJ_SIMP_TAC;
OK..
> val it =
    Initial goal proved.
    |- !r x y s. rect_ex r s ==>
         (rect_get_x r (rect_move r x y s) = rect_get_x r s + x)
      : goalstack
```

その他の定理

上記以外にも次のような定理が証明可能である．

```
|- !f s. fig_ex f s ==>
     (fig_getPos f (fig_sym0 f s) =
       (~fig_get_x f s, ~fig_get_y f s))

|- !r s. rect_ex r s ==>
     (rect_getPos r (rect_sym0 r s) =
       (~rect_get_x r s - rect_get_w r s,
        ~rect_get_y r s - rect_get_h r s))

|- !c s. crect_ex c s ==>
     (crect_getPos c (crect_sym0 c s) =
       (~crect_get_x c s - crect_get_w c s,
        ~crect_get_y c s - crect_get_h c s))

|- !f s. fig_ex f s ==>
     (fig_getPos f (fig_sym0 f (fig_sym0 f s)) = fig_getPos f s)

|- !r s. rect_ex r s ==>
     (rect_getPos r (rect_sym0 r (rect_sym0 r s)) = rect_getPos r s)

|- !c s. crect_ex c s ==>
     (crect_getPos c (crect_sym0 c (crect_sym0 c s)) = crect_getPos c s)

|- !f s. fig_ex f s ==>
     (fig_getPos f (v_fig_sym0 f (v_fig_sym0 f s)) = v_fig_getPos f s)
```

第2章　オブジェクト指向理論

本章では，前章で概略的に示したオブジェクト指向理論の明確な定義を与える．オブジェクト指向理論は，クラスモデルの要素，つまり，クラスや属性，継承関係などを，HOL の型や演算子により表現し，公理を導入することにより定義される．

本章の構成は以下の通りである．まず，2.1 節ではクラスモデルの定義を示す．次に，2.2, 2.3 節ではクラスモデルに基づき理論に導入される型，演算子の定義を示す．2.4 節では公理を示す．

2.1　クラスモデル

クラスモデルは，システムの静的構造を定義する．具体的には，クラス名，属性名とその型，デフォルト値，及び，継承関係を定義する．

クラスモデルを以下の 6 つ組として定義する．

$$CM = (C, A, \mathcal{M}_{attr}, \mathcal{M}_{inher}, \mathcal{T}, \mathcal{V})$$

- $\mathcal{M}_{attr} : C \to Pow(A)$
- $\mathcal{M}_{inher} : C \to Pow(C)$
- $\mathcal{T} : C \times A \to Type$
- $\mathcal{V} : C \times A \to Value$

C，A はそれぞれ，システムに出現するクラス名の集合，属性名の集合である．$\mathcal{M}_{attr}$ は，クラスとその属性集合を対応付ける写像である．$\mathcal{M}_{inher}$ は，クラスと，その子クラス集合を対応付ける写像である．継承関係は，Java のような単一継承であり，木構造をなす．ただし，ルートクラスは 1 つとは限らず，複数の継承木が存在してもよい．$\mathcal{T}$ は，クラスと属性に対し，その属性の型を対応付ける写像である．集合 $Type$ は，HOL における型変数を含まない任意の型の集合である．$c \in C$ と同名の型が $Type$ に存在するとし，クラス名によりそのクラスに属すオブジェクトの型を表す．$\mathcal{V}$ は，クラスと属性に対し，その属性のデフォルト値を対応付ける写像である．集合 $Value$ は $Type$ に含まれるすべての型の要素の集合である．

以下に，いくつかの記号を導入する．

$c \lhd d$ により，c が d の親クラスであることを表す（UML の継承の三角記号から連想）．つまり，$c \lhd d = d \in \mathcal{M}_{inher}(c)$ である．さらに，$c \lhd^{+} d$ は，c が d の祖先クラスであることを，$c \lhd^{*} d$ は，$c = d$ または $c \lhd^{+} d$ であることを意味する．例えば，$\mathtt{fig} \lhd \mathtt{rect}$，$\mathtt{fig} \lhd^{+} \mathtt{crect}$，$\mathtt{fig} \lhd^{*} \mathtt{fig}$ である．

$attr(c)$ により，クラス c の属性と，継承された属性の集合を表す．つまり，$attr(c) = \{a \mid a \in \mathcal{M}_{attr}(d), d \lhd^{*} c\}$ である．例えば，$attr(rect) = \{x, y, w, h\}, attr(crect) = \{x, y, w, h, c\}$ である．

$relative(c, d)$ により，クラス c, d が同一の継承木に属すことを表す．つまり，$relative(c, d) = \exists r.\ r \lhd^{*} c \wedge r \lhd^{*} d$ である．

2.2 型，定数

ストアは，型 $store$ として定義する．$store$ は定数として Emp を持つ．これはどのクラスからもオブジェクトが生成されていない，空のストアを表す．

クラス c に属すオブジェクトは，型 c を持つとする．各 c は，定数として $Null^{c}$ を持つ．これはそのクラスの NULL オブジェクトを表す．

クラス c の属性 a の未定義の値を意味する定数として，$Unknown_{a}^{c} : \mathcal{T}(c, a)$ を導入する．これはストアに存在しないオブジェクトに対し属性取得を行った場合に返される値である．

HOL において，$store$, Emp, $Null^{c}$, $Unknown_{a}^{c}$ はそれぞれ，`store`, `store_emp`, `c_null`, `c_unknown_a` と表記する．

2.3 演算子

6 種類の基本演算子を以下に定義する．

- **new 演算子**

  $$New^{c} : store \rightarrow c * store \quad (c \in C)$$

  クラス c のオブジェクトを生成する関数．$New^{c}\ s = (o, s')$ であるとき，o は新しく生成された c クラスのオブジェクト，s' は生成後のストアである．

- **ex 演算子**

  $$Ex^{c} : c \rightarrow store \rightarrow bool \quad (c \in C)$$

  クラス c のオブジェクトがストアに存在するかどうかを検査する述語．$Ex^{c}\ o\ s = x$ であるとき，x はオブジェクト o がストア s に存在するかどうかの真偽値である．

- **get 演算子**

$$Get^c_a : c \to store \to \mathcal{T}(c,a) \quad (c \in C, a \in attr(c))$$

クラス c **のオブジェクトの属性** a **を取得する関数．** $Get^c_a\ o\ s = x$ **のとき，** x **はオブジェクト** o **の属性** a **のストア** s **における値である．**

- **set 演算子**

$$Set^c_a : c \to \mathcal{T}(c,a) \to store \to store \quad (c \in C, a \in attr(c))$$

クラス c **のオブジェクトの属性** a **を更新する関数．** $Set^c_a\ o\ x\ s = s'$ **のとき，** s' **はストア** s **においてオブジェクト** o **の属性** a **を** x **に更新して得られるストアである．**

- **cast 演算子**

$$Cast^c_d : c \to store \to d \quad (c, d \in C, c \lhd^+ d \text{ or } d \lhd^+ c)$$

クラス c **のオブジェクトをクラス** d **のオブジェクトに型変換する関数．** $Cast^c_d\ o\ s = o'$ **のとき，** o' **は** c **型のオブジェクト** o **をストア** s **において型変換して得られる** d **型のオブジェクトである．**

- **is 演算子**

$$Is^c_d : c \to store \to bool \quad (c, d \in C, c \lhd^* d)$$

クラス c **のオブジェクトがクラス** d **のインスタンスであるかを検査する述語．** $Is^c_d\ o\ s = x$ **のとき，** x **はストア** s **においてオブジェクト** o **がクラス** d **のインスタンスであるかどうかの真偽値である．**

HOL **において，** Ex^c, New^c, Get^c_a, Set^c_a, $Cast^c_d$, Is^c_d **はそれぞれ，** `c_ex`, `c_new`, `c_get_a`, `c_set_a`, `c_cast_d`, `c_is_d` **と表記する．**

2.4 公理

以上で定義した演算子に関する公理を以下に導入する．また，公理と定数，演算子の関係を表 B.1, B.2 **に示す．**

1. **NOT_EX_EMP**

$$\forall o.\ \neg(Ex^c\ o\ Emp)$$

ストアの初期値 Emp **にはオブジェクトは存在しない．**

2. **NOT_EX_NULL**

$$\forall s.\ \neg(Ex^c\ Null^c\ s)$$

NULL **オブジェクトはストアに存在しない．**

3. **EX_IS**

$$\forall o\ s.\ Ex^c\ o\ s = Is^c_{d_1}\ o\ s \vee ... \vee Is^c_{d_n}\ o\ s\ (\{d_1, ..., d_n\} = \{d \mid c \lhd^* d\})$$

ストアに存在するクラス c **のオブジェクト** o **は，**c **または，その子孫クラスのいずれかのインスタンスである．**

4. **NOT_EX_FST_NEW**

$$\forall s.\ \neg(Ex^c\ (Fst\ (New^c\ s))\ s)$$

新しく生成されたオブジェクトは生成前のストア s **に存在しない．**

5. **NOT_EX_FST_NEW_CAST**

$$\forall s.\ \neg(Ex^c\ (Cast^c_d\ (Fst\ (New^c\ s))\ (Snd\ (New^x\ s)))\ s)$$

新しく生成されたオブジェクトを型変換して得られるオブジェクトは，生成前のストア s **に存在しない．**

6. **IS_IMP_NOT_IS**

$$\forall o\ s.\ Is^c_d\ o\ s \Rightarrow \neg(Is^c_e\ o\ s)\quad (d \neq e)$$

クラス c **のオブジェクト** o **がクラス** d **のインスタンスであれば，**d **とは異なるクラス** e **のインスタンスではない．**

7. **IS_CAST**

$$\forall o\ s.\ Is^c_e\ o\ s \Rightarrow Is^d_e\ (Cast^c_d\ o\ s)\ s\quad (d \lhd^+ e)$$

クラス c **のオブジェクト** o **がクラス** e **のインスタンスであれば，**o **を** e **よりも祖先であるいずれのクラス** d **に型変換したとしても，そのオブジェクトは** e **のインスタンスである．つまり，生成時の型は，型変換により見かけの型が変化しても，一定である．**

8. **IS_NEW**

$$\forall o\ s.\ Is_c^c\ o\ (Snd\ (New^c\ s)) = (o = Fst\ (New^c\ s)) \vee Is_c^c\ o\ s$$

クラス c のオブジェクトを生成後，オブジェクト o がクラス c のインスタンスであることは，o がその新しく生成されたオブジェクトであるか，または，生成前のストアにおいてすでに c のインスタンスであったオブジェクトであることと同値である．

9. **IS_NEW_CAST**

$$\forall o\ s.\ Is_d^c\ o\ (\#2(New^d\ s)) =$$
$$(o = Cast_c^d\ (Fst\ (New^d\ s))\ (Snd\ (New^d\ s)) \vee Is_d^c\ o\ s$$

クラス d のオブジェクトを生成後，d の祖先クラス c のオブジェクト o が d のインスタンスであることは，o がその新しく生成されたオブジェクトを c に型変換して得られたオブジェクトであるか，または，生成前のストアにおいてすでに d のインスタンスであった c オブジェクトであることと同値である．

10. **DIFF_IS_NEW**

$$\forall o\ s.\ Is_d^c\ o\ (Snd\ (New^e\ s)) = Is_d^c\ o\ s \quad (d \neq e)$$

クラス c のオブジェクトがクラス d のインスタンスであることは，d とは異なるクラス e のオブジェクトの生成には無関係である．

11. **IS_SET**

$$\forall o_1\ o_2\ x\ s.\ Is_d^c\ o_1\ (Set_a^e\ o_2\ x\ s) = Is_d^c\ o_1\ s$$

あるオブジェクトがどのクラスのインスタンスであるかは，属性の更新には無関係である．

12. **DOWN_NULL**

$$\forall o\ s.\ Is_c^c\ o\ s \Rightarrow (Cast_d^c\ o\ s = Null^d) \quad (c \lhd^+ d)$$

クラス c のインスタンスをそれよりも子孫のクラス d に型変換すれば，変換先のクラスの NULL オブジェクトとなる．

13. **NOT_EX_CAST**

$$\forall o\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Cast_d^c\ o\ s = Null^d)$$

ストアに存在しないオブジェクトを型変換すれば変換先のクラスの NULL オブジェ

クトとなる．

14. **UP_11**

$$\forall o_1\ o_2\ s.\ Ex^d\ o_1\ s \wedge Ex^d\ o_2\ s \Rightarrow$$
$$\neg(o_1 = o_2) \Rightarrow \neg(Cast^d_c\ o_1\ s = Cast^d_c\ o_2\ s)\ \ (c \lhd^+ d)$$

ストアに存在するクラス d の２つのオブジェクト o_1，o_2 について，それらが異なるオブジェクトならば，それらを祖先クラス c に型変換しても異なるオブジェクトとなる．

15. **DOWN_11**

$$\forall o_1\ o_2\ s.\ Is^c_e\ o_1\ s \wedge Is^c_e\ o_2\ s \Rightarrow$$
$$\neg(o_1 = o_2) \Rightarrow \neg(Cast^c_d\ o_1\ s = Cast^c_d\ o_2\ s)\ \ (c \lhd^+ d \text{ and } d \lhd^* e)$$

クラス e のインスタンスであるクラス c の２つのオブジェクト o_1，o_2 について，それらが異なるオブジェクトならば，それらを c の子孫であり e の祖先であるクラス d に型変換しても異なるオブジェクトとなる．

16. **UP_DOWN**

$$\forall o\ s.\ Ex^d\ o\ s \Rightarrow (Cast^c_d\ (Cast^d_c\ o\ s)\ s = o)\ \ (c \lhd^+ d)$$

クラス d のオブジェクト o がストアに存在するならば，それを祖先クラス c に型変換し，もう一度，d に型変換すれば，もとのオブジェクト o 自身となる．

17. **DOWN_UP**

$$\forall o\ s.\ Ex^d\ (Cast^c_d\ o\ s)\ s \Rightarrow (Cast^d_c\ (Cast^c_d\ o\ s)\ s = o)\ \ (c \lhd^+ d)$$

クラス c のオブジェクト o を子孫クラス d に型変換可能（型変換して得られるオブジェクトがストアに存在する）ならば，それを子孫クラス d に型変換し，もう一度，c に型変換すれば，もとのオブジェクト o 自身となる．なお，o を c から d に型変換可能であるのは，o が d または，d よりも子孫のクラスのインスタンスである．

18. **CAST_CAST**

$$\forall o\ s.\ Cast^d_e\ (Cast^c_d\ o\ s)\ s = Cast^c_e\ o\ s$$
$$((c \lhd^+ d \text{ and } d \lhd^+ e) \text{ or } (e \lhd^+ d \text{ and } d \lhd^+ c))$$

これは cast 演算子の推移的な適用に関する公理である．クラス c からクラス e に型

変換することは，c から一旦，継承関係において c と e の間にあるクラス d に型変換し，それを d に型変換することに等しい．

19. **CAST_SET**

$$\forall o_1\ o_2.\ Cast^c_d\ o_1\ (Set^e_a\ o_2\ x\ s) = Cast^c_d\ o_1\ s$$

型変換は属性の更新に無関係である．

20. **EX_CAST_NEW**

$$\forall o\ s.\ Ex^c\ o\ s \Rightarrow\ (Cast^c_d\ o\ (Snd\ (New^e\ s)) = Cast^c_d\ o\ s)$$
$$(c \lhd^* e \text{ and } d \lhd^* e)$$

クラス c, d がともにクラス e の子孫であるとき，c のオブジェクト o を，d へ型変換して得られるオブジェクト参照の値は，o がストアに存在するならば，e インスタンスの生成に無関係である．つまり，型変換の結果得られるオブジェクトは，すでに存在するオブジェクトであるため，新しく生成される e インスタンスを型変換して得られるオブジェクトとは異なるオブジェクトである．

21. **DIFF_CAST_NEW**

$$\forall o\ s.\ Cast^c_d\ o\ (Snd\ (New^e\ s)) = Cast^c_d\ o\ s\ \ (c \not\lhd^* e \text{ or } d \not\lhd^* e)$$

あるオブジェクト o を，クラス e のインスタンスの型変換の有効範囲（e の子孫クラス間での変換）を越えるクラス間で型変換を行った場合，得られるオブジェクト参照の値は，e インスタンスの生成に無関係である．つまり，型変換の結果得られるオブジェクトは，e インスタンスを型変換して得られるオブジェクトとは異なるオブジェクトである．

22. **NOT_EX_GET**

$$\forall o\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Get^c_a\ o\ s = Unknown^d_a)\ \ (a \in \mathcal{M}_{attr}(c))$$

ストアに存在しないオブジェクトの属性を取得した場合，その値は，その属性の未定義の値を意味する定数となる．

23. **NOT_EX_SET**

$$\forall o\ x\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Set^c_a\ o\ x\ s = s)$$

ストアに存在しないオブジェクトの属性更新は無効となる．

24. **SPR_GET**

$$\forall o\ s.\ Get_a^d\ o\ s = Get_a^c\ (Cast_c^d\ o\ s)\ s \quad (c \lhd^+ d \text{ and } a \in attr(c))$$

クラス c が属性 a を持つとき，c の子孫クラス d のオブジェクト o の属性 a を取得して得られる値は，o を c に型変換し，そのオブジェクトについて得られる a の値と等しい．

25. **SPR_SET**

$$\forall o\ s.\ Set_a^d\ o\ x\ s = Set_a^c\ (Cast_c^d\ o\ s)\ x\ s \quad (c \lhd^+ d \text{ and } a \in attr(c))$$

クラス c が属性 a を持つとき，c の子孫クラス d のオブジェクト o の属性 a を更新することは，o を c に型変換し，そのオブジェクトについて a を更新することに等しい．

26. **GET_SET**

$$\forall o\ s.\ Ex^c\ o\ s \Rightarrow (Get_a^c\ o\ (Set_a^c\ o\ x\ s) = x)$$

オブジェクト o がストアに存在するならば，その属性 a を x に更新し，その直後に同じ属性を取得したとき，その値は x である．

27. **DIFF_OBJ_GET_SET**

$$\forall o_1\ o_2\ s.\ \neg(o_1 = o_2) \Rightarrow (Get_a^c\ o_1\ (Set_a^c\ o_2\ x\ s) = Get_a^c\ o_1\ s)$$

オブジェクト o_2 の属性 a を更新直後，それとは異なるオブジェクト o_1 の同じ属性を取得したとき，その値は，更新前に得られる値と等しい．つまり，異なる２つのオブジェクトの属性は独立している．

28. **DIFF_GET_SET**

$$\forall o_1\ o_2\ s.\ Get_a^c\ o_1\ (Set_b^d\ o_2\ x\ s) = Get_a^c\ o_1\ s$$
$$((c \not\lhd^* d \text{ and } d \not\lhd^* c) \text{ or } a \neq b)$$

クラス c, d が継承関係にないとき，または，属性 a, b が異なるとき，d のオブジェクト o_2 の属性 b を更新直後，c のオブジェクト o_1 の属性 a を取得したとき，その値は，更新前に得られる値と等しい．これも??と同様であり，異なる２つの属性は独立していることを意味する．

29. **GET_NEW**

$$\forall s.\ Get^c_a\ (Fst\ (New^c\ s))\ (Snd\ (New^c\ s)) = \mathcal{V}(d,a)\quad (a \in \mathcal{M}_{attr}(d))$$

オブジェクトを生成直後，その新しいオブジェクトの属性を取得した場合，その値はその属性のデフォルト値となる．

30. **GET_NEW_CAST**

$$\forall o\ s.\ (o = Cast^d_c\ (Fst\ (New^d\ s))\ (Snd\ (New^d\ s))) \Rightarrow$$
$$(Get^c_a\ o\ (Snd\ (New^d\ s)) = \mathcal{V}(e,a))\quad (c \lhd^+ d \text{ and } a \in \mathcal{M}_{attr}(e))$$

オブジェクトを生成直後，その新しいオブジェクトを祖先クラスに型変換し，属性を取得した場合，その値はその属性のデフォルト値となる．

31. **EX_GET_NEW**

$$\forall o\ s.\ Ex^c\ o\ s \Rightarrow (Get^c_a\ o\ (Snd\ (New^d\ s)) = Get^c_a\ o\ s)\quad (c \lhd^* d)$$

オブジェクト生成直後，ストアにすでに存在するオブジェクトの属性を取得する場合，その値は生成前に取得する値と同一となる．つまり，o が存在すれば，o は新しく生成されるオブジェクトとは異なるオブジェクトであるから，その属性更新はオブジェクト生成とは無関係に行うことができる．

32. **DIFF_GET_NEW**

$$\forall o\ s.\ Get^c_a\ o\ (Snd\ (New^d\ s)) = Get^c_a\ o\ s\quad (c \not\lhd^* d)$$

クラス d のオブジェクト生成直後，d の祖先ではないクラス c のオブジェクト o の属性を取得する場合，その値は生成前に取得する値と同一となる．つまり，c は d の祖先ではないため，o は新しく生成されるオブジェクトとは異なるオブジェクトであり（d インスタンスは型変換によって c オブジェクトにはなりえない），属性取得はオブジェクト生成の影響を受けない．

33. **SET_SET**

$$\forall o\ x\ y\ s.\ Set^c_a\ o\ x\ (Set^c_a\ o\ y\ s) = Set^c_a\ o\ x\ s$$

あるオブジェクトについて，属性 a を更新直後，同じ属性を更新する場合，前の更新は無効となる．

34. **DIFF_OBJ_SET_SET**

$$\forall o_1\ o_2\ x\ y\ s.\ \neg(o_1 = o_2) \Rightarrow$$
$$(Set^c_a\ o_1\ x\ (Set^c_a\ o_2\ y\ s) = Set^c_a\ o_2\ y\ (Set^c_a\ o_1\ x\ s))$$

オブジェクト o_1 の属性 a の更新と，それとは異なるオブジェクト o_2 の属性 a の更新について，その更新順序は入れ替え可能である．

35. **DIFF_SET_SET**

$$\forall o_1\ o_2\ x\ y\ s.\ Set^c_a\ o_1\ x\ (Set^d_b\ o_2\ y\ s) = Set^d_b\ o_2\ y\ (Set^c_a\ o_1\ x\ s)$$
$$((c \not\lhd^* d \text{ and } d \not\lhd^* c) \text{ or } (a \neq b))$$

属性 a の更新と，a とは異なる属性 b の更新について，その更新順序は入れ替え可能である．

36. **EX_SET_NEW**

$$\forall o\ x\ s.\ Ex^c\ o\ s \Rightarrow$$
$$(Set^c_a\ o\ x\ (Snd\ (New^d\ s)) = Snd\ (New^d\ (Set^c_a\ o\ x\ s)))\ \ (c \lhd^* d)$$

クラス c のオブジェクト o の属性の更新と，c の子孫クラス d のオブジェクトの生成は，o がストアに存在すれば，その順序は入れ替え可能である．つまり，o が存在すれば，o は新しく生成されるオブジェクトとは異なるオブジェクトであるから，その属性更新はオブジェクト生成とは無関係に行うことができる．

37. **DIFF_SET_NEW**

$$\forall o\ x\ s.\ Set^c_a\ o\ x\ (Snd\ (New^d\ s)) = Snd\ (New^d\ (Set^c_a\ o\ x\ s))\ \ (c \not\lhd^* d)$$

クラス c のオブジェクト o の属性の更新と，c の子孫ではないクラス d のオブジェクトの生成について，その順序は入れ替え可能である．つまり，c は d の祖先ではないため，o は新しく生成されるオブジェクトとは異なるオブジェクトであり（d インスタンスは型変換によって c オブジェクトにはなりえない），その属性更新はオブジェクト生成とは無関係に行うことができる．

38. **DIFF_NEW_NEW**

$$\forall s.\ Snd\ (New^c\ (Snd\ (New^d\ s))) = Snd\ (New^d\ (Snd\ (New^d\ s)))$$
$$(\text{not } relative(c, d))$$

異なる継承木に属すクラス c, d のオブジェクトの生成順序は入れ替え可能である．

39. **SET_GET**

$$\forall o\ s.\ Set_a^c\ o\ (Get_a^c\ o\ s)\ s = s$$

ある属性をそのストアにおける値で更新すれば，もとのストアに一致する．

40. **FST_NEW_SET**

$$\forall o\ x\ s.\ Fst\ (New^c\ (Set_a^d\ o\ x\ s)) = Fst\ (New^c\ s)$$

オブジェクト生成によって得られる新しいオブジェクト参照の値は属性更新の影響を受けない．

41. **DIFF_FST_NEW_NEW**

$$\forall s.\ Fst\ (New^c\ (Snd\ (New^d\ s))) = Fst\ (New^c\ s)\ \ (c \not\trianglelefteq^* d)$$

クラス c のオブジェクト生成によって得られる新しいオブジェクト参照の値は，c の子孫ではないクラス d のオブジェクト生成の影響を受けない．

第3章　オブジェクト指向理論の実装

本章では，HOL におけるオブジェクト指向理論の実装について述べる．本章の構成は以下の通りである．まず，3.1 においてオブジェクト指向理論構築の背景を述べる．次に，3.2 において実装の概要を例を用いて述べる．最後に，3.3 節において一般的な実装法を述べる．

3.1　背景

現在，実用的に使われている定理証明器は高階述語論理に基づくものであり，代表的なものには，Isabelle/HOL, PVS, HOL, Coq などがある．これらを用いてオブジェクト指向プログラムやモデルの検証を行う場合，その前段階として，クラスや属性，継承などのオブジェクト指向概念を高階述語論理の上に実装することが必要となる．

実際，これまで，多くのオブジェクト指向理論が高階述語論理の定理証明器に実装されてきた．その多くは Java プログラム検証の意味論として実装されたものである．有力な研究には，LOOP(Isabelle/HOL, PVS)[7]，Bali(Isabelle/HOL)[9]，Krakatoa(Coq)[10] などがある．

我々の検証対象はプログラムではなく，分析モデルである．プログラムと分析モデルの大きな違いは出現する型の多様性である．図 3.1 に示すように，Java に出現する型は，int, bool, array といった基本的なものに限られるが，分析モデルに出現する型はより広範であり，set, bag, stack のような抽象型や，date, time, currency などの対象領域に特化した型が出現する．これまでの Java プログラム検証のための理論では，Java に出現する特定の型しか扱うことができなかったが，我々のオブジェクト指向理論においては，分析モデルに出現しうる任意の型を扱えるように工夫をしている．

任意の型を扱えるオブジェクト指向理論を実装する際の問題点として，HOL の単純一階型システムにおいてどのように任意の型を持ちうるオブジェクトを表現するか，という問題がある．一般に，オブジェクトは，任意の型の属性を任意個結合した型と考えることができる．このような型は，一見，複数の型変数の直積型として $(\alpha * \beta * \gamma * ...)$ のように素直に表現可能と考えられる．しかし，オブジェクトは任意個の属性を持ちうるため，いくつの要素に対して直積をとればよいかが予め分からない．レコード型や直和型を用いても同様の問題が生じる．これを解決する方法としては，extensible record という概念を用いる手法が提案されているが，オブジェクトに参照としての性質を持たせるまでには至っていない．このようなことから，オブジェクト指向の概念を扱うためには HOL よりも高度な型システムの開発が必要であるいわれており，object calculus[5] のようなオブジェク

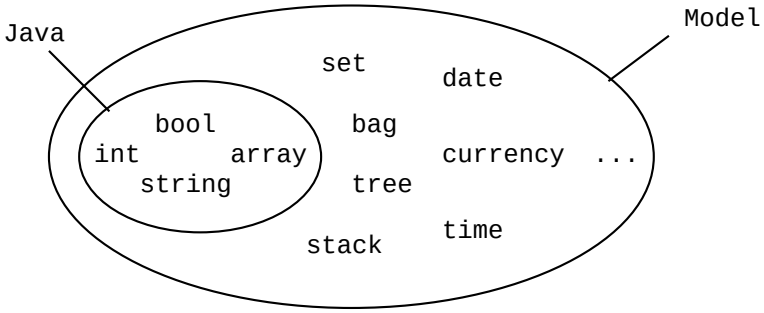

図 3.1: プログラムと分析モデルの違い

トを専門に扱うための型システムの研究が幅広く行われている．しかし，現時点では，そのような型システムを標準的に実装している定理証明器は存在しないため，既存の HOL の型システムになんらかの工夫をしてオブジェクト指向概念を実装しなければならない．

この問題に対処する方法として，我々は，オブジェクト指向理論を対象システムの型情報が含まれるクラスモデルをもとに自動生成する，というアプローチをとっている．つまり，オブジェクトを一般的な型で表現するかわりに，対象システムに特化した型として自動生成するというアプローチである．基本的な発想は，属性の型が入力として予め与えられれば，それを格納するオブジェクトは $(num * int * bool)$ のように直積により簡単に表現できるということである．

本検証ツールでは，クラスモデルからそのモデルに特化したオブジェクト指向理論を自動生成している．理論は definitional extension により構築している．これは，conservative extension とも呼ばれ，既存の健全な理論から，定義の導入または健全な推論規則による導出のみを許して新しい理論を構築していく手法である．この手法により構築された理論は健全性が保証される．本検証ツールは，HOL の既存の自然数，ペア，リストといった理論によって，オブジェクトを格納するためのヒープメモリ構造を定義し，その操作的意味論からオブジェクト指向理論を導出している．

3.2 概要

オブジェクト指向理論の構築は，まず，オブジェクトを格納するためのヒープメモリ構造を自然数やリストといった基礎的な理論によって定義し，次に，その操作的意味論から公理を導出するという方法で行っている．

まず，ストアの構造について説明する．ストアはオブジェクトのデータを格納するためのヒープメモリとして実装される．図 3.2 は，図 1.3 の例におけるヒープメモリのスナップショットである．ヒープメモリ全体は，3 つのクラス `fig`，`rect`，`crect` に対応する 3 つのサブヒープから構成される．それぞれのサブヒープは，リストにより表現され，ヒープメモリ全体は３つのリストの組として表現される．

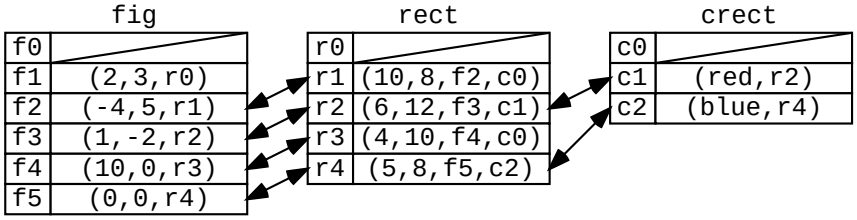

図 3.2: ヒープメモリの表現

各クラスのオブジェクト参照は，そのクラスに対応するサブヒープのインデックスで表現される．クラス fig に対応するサブヒープの場合，fig 型の参照 f1，f2，... は，それぞれ，自然数 1，2，... に対応している．f0 は NULL 参照 fig_null として使用される．オブジェクトの実体は，各クラスのサブヒープに格納される複数のセルにより表現される．例えば，セル f1 は，属性値が x=2，y=3 である fig クラスのインスタンスを表す．2 つのセル f2，r1 は，属性値が x=-4，y=5，w=10，h=8 である rect クラスのインスタンスを表す．3 つのセル f3，r2，c1 は，属性値が x=1，y=-2，w=6，h=12，c=red である crect クラスのインスタンスを表す．1 つのオブジェクトを構成する複数のセルは，オブジェクト参照を互いに格納することによりリンクされる．1 つの rect インスタンスを構成する 2 つのセル f2，r1 は，参照 r1，f2 をそれぞれ格納することによりリンクしている．リンクするセルが存在しないときは，NULL 参照が格納される．セル f1 は，どの rect セルともリンクしないので，r0 を格納しているこのように，複数のセルから一つのオブジェクトを構成するメカニズムにより，オブジェクトのサブタイピングが実現される．例えば，3 つの参照 f3，r2，c1 はいずれも同じ crect インスタンスを指しているが，これは，このインスタンスが 3 つの見かけの型をとることができることを意味する．

次に，6 つの基本演算子の実装について説明する．new 演算子は，各クラスのメモリに新しいセルを追加し，それらを互いにリンクする関数として実装される．例えば，rect_new は，fig クラスのサブヒープ，rect クラスのサブヒープにそれぞれ 1 つづつセルを追加し，それらを互いにリンクする関数である．セルを追加する際，セルの各要素にはデフォルト値を格納する．ex 演算子は，参照がサブヒープの有効な範囲を指しているかを検査する述語として実装される．例えば，fig_ex は，fig クラスに対応するサブヒープにおいて，第一引数の fig 参照が，0 でなく，さらに，サブヒープの長さ以上でないことを検査する述語である．cast 演算子は，参照から，親クラスまたは子クラスの参照へセルを辿る関数として実装される．例えば，fig_cast_rect は，第一引数の fig 参照が指すセルを取得し，それに格納される rect 参照を取得する関数である．get 演算子，set 演算子は，参照が指すセルに格納される属性を，それぞれ取得，更新する関数として実装される．is 演算子は，参照が指すセルのリンク構造に基づき，それがどのクラスのインスタンスであるか判別する述語として実装される．セルのリンク構造に着目すれば，そのクラスを一意に判別することができる．例えば fig 参照の場合，それが，f1 のように fig クラスのインスタンスを指しているならば，fig セルから rect セルへのリンクは NULL である．また，

f2 のように rect クラスのインスタンスを指しているならば，fig セルから rect セルへのリンクは存在するが，rect セルから crect セルへのリンクは NULL である．f3 のように crect クラスのインスタンスを指しているならば，fig セルから rect セルへのリンク，rect セルから crect セルへのリンク両方が存在する．このように，セルのリンクがサブクラスの方向にどこまで辿れるかに注目すれば，fig 参照がどのクラスのインスタンスを指しているか識別可能である．is 演算子は，このようなリンク構造に基づいて実装される．

すべての公理はこれらの関数，述語の定義から導出可能である．例えば，公理 26 や 27 は関数定義の展開と自然数やリストに関する定理による書き換えのみによって簡単に証明することができる．公理 14 や 16 は特殊であり，ヒープメモリ構造の不変表明として証明される．不変表明の証明には，ヒープメモリに対する構造帰納法を用いる．つまり，初期段階として，ヒープメモリの初期値について成立することを証明し，帰納段階として，ヒープメモリに対する書き込み演算とオブジェクト生成演算の適用前後で成立することを証明する．

ML においても同様の構造を持つヒープメモリを自動生成することが可能である．これによりオブジェクト指向理論のシミュレーション実行が可能となる．

3.3 ヒープメモリの実装

3.3.1 サブヒープのデータ構造

サブヒープは，クラスモデルに依存せず，一般的にリスト $'a\ list$ として表現される．データのアドレスはリストのインデックス，つまり，自然数 0, 1, 2,... で表現する．サブヒープの初期値を $[null]$ とする．これは，アドレス 0 に NULL データを表すダミーの定数 $null : 'a$ を格納したリストである．サブヒープ上の演算子として，$add, valid, read, write$ の 4 つを図 3.3 に定義する．

add は，リストの最後尾にデータ x を追加し，そのアドレスと追加後のリストを返す関数である．$valid$ は，指定されたアドレス n に生成されたデータが存在するかを検査する述語である．アドレスが有効である範囲は，0 より大きく，現在のリスト長より小さい範囲である．$read$ は，指定されたアドレス n のデータを読みだす関数である．アドレスが無効である場合は，$unknown$ という未定義の値を意味する定数を返す．$write$ は，指定されたアドレス n にデータ x を書き込む関数である．アドレスが無効である場合は，ヒープに変更を加えない．

3.3.2 オブジェクト参照の表現

オブジェクト参照の型は，サブヒープのインデックス，つまり，自然数との間に全単射をとることにより得られる．クラス c のオブジェクト参照の型は次のように定義される．

$$
\begin{array}{l}
add : 'a \rightarrow 'a\ list \rightarrow 'a\ list \\
add\ x\ l \equiv (Length\ l, Append\ l\ [x]) \\
\\
valid : num \rightarrow 'a\ list \rightarrow bool \\
valid\ n\ l \equiv (0 < n) \wedge (n < length\ l) \\
\\
read : num \rightarrow 'a\ list \rightarrow 'a \\
read\ n\ l \equiv if\ valid\ n\ l\ then\ read_1\ n\ l\ else\ unknown \\
\quad where \\
\quad\quad (read_1\ 0\ l = Hd\ l) \wedge (read_1\ (Suc\ n)\ l = read_1\ n\ (Tl\ l)) \\
\\
write : num \rightarrow 'a \rightarrow 'a\ list \rightarrow 'a\ list \\
write\ n\ x\ l \equiv if\ valid\ n\ l\ then\ write_1\ n\ x\ l\ else\ l \\
\quad where \\
\quad\quad (write_1\ 0\ x\ l = x :: (Tl\ l)) \wedge \\
\quad\quad (write_1\ (Suc\ n)\ x\ l = (Hd\ l) :: (write_1\ n\ x\ (Tl\ l)))
\end{array}
$$

図 3.3: **サブヒープ上の演算子**

$$
\begin{array}{l}
HOL_datatype\ c = AbsObj_c\ of\ num \\
RepObj_c\ (AbsObj_c\ n) \equiv n
\end{array}
$$

関数 $AbsObj_c$ **は自然数から** c **オブジェクトへの写像，関数** $RepObj_c$ **は** c **オブジェクトから自然数への写像である．**

NULL **オブジェクトは** 0 **により表現する．つまり，**

$$
Null^c \equiv AbsObj_c\ 0
$$

3.3.3 ヒープメモリのデータ構造

サブヒープはクラスモデルに存在する各クラスに対応して導入され，それぞれ異なる型のタプルが格納される．クラス c **に対応するサブヒープに格納されるタプルの型を次のように定義する．**

$$tuple_c \equiv attrs_c * d * e_1 * ... * e_m \quad (d \lhd c, c \lhd e_j)$$
$$where$$
$$attrs_c \equiv \mathcal{T}(c, a_1) * ... * \mathcal{T}(c, a_n) \quad (a_i \in \mathcal{M}_{attr}(c))$$

タプルの最初の要素 $attrs_c$ はクラス c で定義される属性をタプルでまとめたものである．次の要素 d は親クラスのオブジェクト参照であり，最後の m 個の要素 $e_1, ..., e_m$ は子クラスのオブジェクト参照である．オブジェクト参照を格納することによりタプル同士の連結を行う．このようなタプルを格納するクラス c のサブヒープの型を次のように定義する．

$$heap_c \equiv tuple_c \ list$$

ヒープメモリ全体は各クラスに導入されたサブヒープをタプルで結合することにより得られる．ヒープメモリの型を次のように定義する．

$$Heap \equiv heap_{c_1} * \ ... \ * heap_{c_n} \quad (c_i \in C)$$

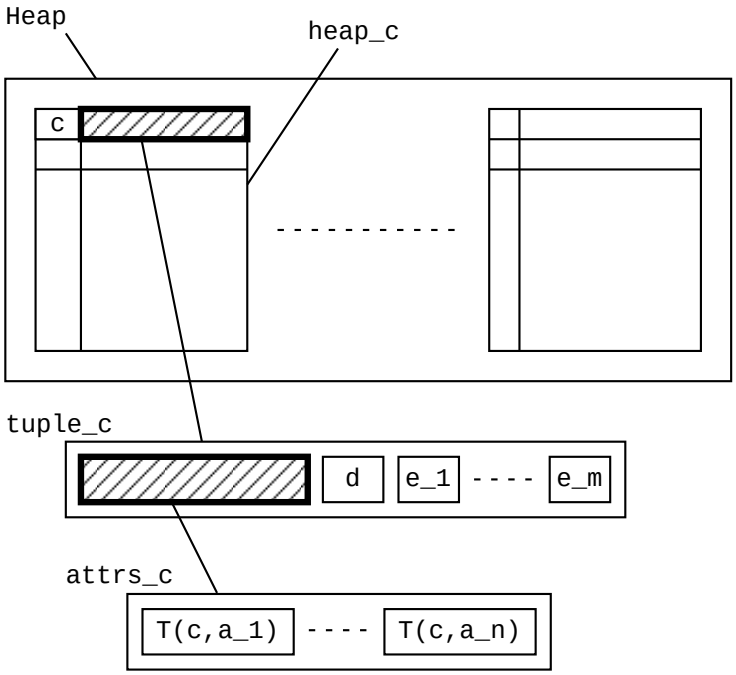

図 3.4: ヒープメモリのデータ構造

ヒープメモリに対する操作として，4 つの演算子 Add^c, $Valid^c$, $Read^c_u$, $Write^c_u$ を図 3.5 に定義する．$Valid^c$ はオブジェクト参照がクラス c に対応するサブヒープにおいて有効であるかどうかを検査する述語である．Add^c はクラス c に対応するサブヒープに新しいタ

プルを追加する関数である．$Read^c_u, Write^c_u$ はオブジェクト参照が指すタプルの要素 u に対し，それぞれ読み出し，書き込みを行う関数である．ただし，$u \in \{a, d, e_1, ..., e_m\}$ であり，$u = a$ のときは属性集合に対する読み書きであり $T = attr_c$，$u = d$ のときは親クラス参照に対する読み書きであり $T = d\ (d \lhd c)$，$u = e_j$ のときは子クラス参照に対する読み書きであり $T = e_j\ (c \lhd e_j)$ である．

$$
\begin{array}{l}
Add^c : tuple^c \rightarrow Heap \rightarrow c * Heap \\
Add^c\ x\ H \equiv \\
\quad let\ (n, l) = add\ x\ (getElem^i_N\ H)\ in \\
\qquad (AbsObj^c\ n, setElem^i_N\ l\ H) \\
\\
Valid^c : c \rightarrow Heap \rightarrow bool \\
Valid^c\ o\ H \equiv valid\ (RepObj^c\ o)\ (getElem^i_N\ H) \\
\\
Read^c_u : c \rightarrow Heap \rightarrow T \\
Read^c_u\ o\ H \equiv \\
\quad if\ Valid^c\ o\ H\ then\ getElem^j_M\ (Read^c\ o\ H)\ else\ (unknown : T) \\
\ \ where \\
\quad Read^c\ o\ H \equiv (read\ (RepObj^c\ o)\ (getElem^i_N\ H)) \\
\\
Write^c_u : c \rightarrow T \rightarrow Heap \rightarrow Heap \\
Write^c_u\ o\ x\ H \equiv \\
\quad if\ Valid^c\ o\ H\ then \\
\qquad let\ t = setElem^j_M\ x\ (Read^c\ (RepObj^c\ o)\ H)\ in \\
\qquad\quad Write^c\ (RepObj^c\ o)\ t\ H \\
\quad else\ H \\
\ \ where \\
\quad Write^c\ o\ t\ H \equiv \\
\qquad let\ l = write\ (RepObj^c\ o)\ t\ (getElem^i_N\ H)\ in \\
\qquad\quad setElem^i_N\ l\ H
\end{array}
$$

図 3.5: ヒープメモリ上の演算子

ここで，$getElem^p_q$, $setElem^p_q$ は，q 個の要素を持つタプルの p 番目の要素を取得，更新する関数である．M はクラス集合 C の要素数であり，i はクラス c に対応するサブヒープの，ヒープメモリにおける位置である．N はクラス c に対応するサブヒープに格納されるタプルの要素数であり，j はそのタプルにおける u の位置である．$getElem^p_q$, $setElem^p_q$ は，p, q に依存して次のように定義される．

$$getElem_q^p\ t = \begin{cases} t & (p=1, q=1\ \text{のとき}) \\ Fst\ t & (p=1, 1<q\ \text{のとき}) \\ Fst\ (Snd_{(1)}\ (...(Snd_{(p-1)}\ t))) & (1<p<q\ \text{のとき}) \\ Snd_{(1)}\ (...(Snd_{(q-1)}\ t)) & (p=q, 1<q\ \text{のとき}) \end{cases}$$

$$setElem_q^p\ x\ t = \begin{cases} x & (q=1\ \text{のとき}) \\ (getElem_q^1\ t, ..., getElem_q^{p-1}\ t, & \\ \quad x, getElem_q^{p+1}\ t, ..., getElem_q^q\ t) & (1<q\ \text{のとき}) \end{cases}$$

3.3.4　ヒープメモリにおけるオブジェクト指向演算子の表現

ヒープメモリ上に定義した演算子を用いて，オブジェクト指向理論の定数 Emp，演算子 Ex^c, $Cast_d^c$, Get^c, Set^c, New^c, Is_d^c の $Heap$ における表現 $EmpRep$, $CastRep_d^c$, $GetRep^c$, $SetRep^c$, $NewRep^c$, $IsRep_d^c$ を定義する．ここで，Get^c, Set^c は，クラス c に定義される複数の属性を同時に取得，更新する関数であるとする．つまり，クラス c は複数の属性をまとめて 1 つのタプルとした属性を唯一持つと考え，Get^c, Set^c はそれを取得，更新する関数である．Get_a^c, Set_a^c はこれらの関数を用いて定義可能である．このような関数を導入するのは，後でヒープメモリのサブセットをストアに抽象する際に，そのサブセットを定義する述語における帰納段階を減らすことができ，証明の効率があがるからである．

$EmpRep$ は次のように定義される．

$$EmpRep : Heap$$
$$EmpRep \equiv ([null : tuple_{c_1}], ..., [null : tuple_{c_n}]) \quad (c_i \in C)$$

空のストアは，空のサブヒープをタプルにまとめたものとして表現される．

$ExRep^c$ は次のように定義される．

$$ExRep^c : c \rightarrow Heap \rightarrow bool$$
$$ExRep^c\ o\ H \equiv Valid^c\ o\ H$$

これは $Valid^c$ そのものである．つまり，クラス c のオブジェクトがストアに存在することはその参照がクラス c に対応するサブヒープにおいて有効であること同値である．

$GetRep^c$, $SetRep^c$ はそれぞれ次のように定義される．

$$GetRep^c : c \rightarrow H \rightarrow attr_c$$
$$GetRep^c\ o\ H \equiv Read_a^c\ o\ H$$

$$SetRep^c : c \rightarrow attr_c \rightarrow H \rightarrow H$$
$$SetRep^c\ o\ x\ H \equiv Write^c_a\ o\ x\ H$$

クラス c のオブジェクトの属性の取得，更新は，そのオブジェクト参照が指すタプルの属性領域の取得，更新と同値である．

$CastRep^c_d$ は次のように定義される．

$$CastRep^c_d : c \rightarrow Heap \rightarrow d$$
$$CastRep^c_d\ o\ H \equiv \begin{cases} if\ ExRep^c\ o\ H\ then\ Read^c_d\ o\ H\ else\ Null^d & (c \lhd d \text{ or } d \lhd c) \\ CastRep^e_d\ (CastRep^c_e\ o\ H)\ H & ((c \lhd e, e \lhd^+ d) \text{ or } (d \lhd^+ e, e \lhd c)) \end{cases}$$

$CastRep^c_d$ はクラス c と d の継承関係の位置について 2 つの場合に分けて定義される．変換元のクラス c と変換先のクラス d が直接の親子関係にある場合は，$Read^c_d$ により，オブジェクト参照が指すブロックに格納されるクラス d のオブジェクト参照を読み出せばよい．変換を行うオブジェクトが存在しない場合は変換先のクラスの NULL オブジェクト $Null^d$ に変換したこととする．c と d が直接の親子関係ではない継承関係にある場合は，型変換を推移的に適用すればよい．つまり，まず c と直接親子関係にあるクラス e に変換してから次にそれを d に変換する．

$NewRep^c$ は次のように定義される．

$$NewRep^c\ H \equiv \begin{cases} Add^c\ default_c\ H & (c\text{がルートのとき}) \\ let\ (o_1, H_1) = NewRep^d\ H\ in & \\ \quad let\ (o_2, H_2) = Add^c\ default_c\ H_1\ in & \\ \quad\quad let\ H_3 = Link^d_c\ o_1\ o_2\ H_2\ in\ \ (o_2, H_3) & (d \lhd c) \end{cases}$$
$$where$$
$$Link^d_c\ o_1\ o_2\ H \equiv Write^c_d\ o_2\ o_1\ (Write^d_c\ o_1\ o_2\ H)$$
$$default_c \equiv ((\mathcal{V}(c, a_1), ..., \mathcal{V}(c, a_n)), Null^d, Null^{e_1}, ..., Null^{e_m})$$
$$(a_i \in \mathcal{M}_{attr}(c), d \lhd c, c \lhd e_j)$$

$NewRep^c$ はクラス c のオブジェクトを，ルートクラスからクラス c に至るまでの継承関係に関して再帰的に構築する．生成されたオブジェクトは複数のタプルが連結した構造となる．まず，初期段階としてルートクラスのインスタンスを生成する場合，Add_c により，クラス c のサブヒープに新たなタプルを追加する．次に，帰納段階としてルート以外のクラスのインスタンスを生成する場合，まず，$NewRep^d$ により親クラス d のインスタンスを生成する．次に，Add^c により，クラス c のサブヒープに新たなタプルを追加し，最後に，得られた二つのオブジェクト参照 o_1, o_2 を $Link^c_d$ によりリンクする．$Link^c_d$ は，$Write^d_c$, $Write^c_d$ によって，o_1, o_2 それぞれが指すタプルのオブジェクト参照の要素に o_2, o_1 を書

き込む関数として定義される．なお，クラス c のサブヒープに追加するタプル $default_c$ の各要素は，属性 a の場合はそのデフォルト値のタプル $(\mathcal{V}(c,a_1),...,\mathcal{V}(c,a_n))$，親クラス d，子クラス e_j のオブジェクト参照の場合はそのクラスの NULL オブジェクト $Null^d$, $Null^{e_j}$ とする．

$IsRep^c_d$ は次のように定義される．

$$
\begin{array}{l}
IsRep^c_d\ o\ H \equiv \\
\begin{cases}
ExRep^c\ o\ H \wedge \bigwedge_j \neg ExRep^{e_j}\ (CastRep^c_{e_j}\ o\ H)\ H & (c = d, c \lhd e_j) \\
ExRep^c\ o\ H \wedge IsRep^e_d\ (CastRep^c_e\ o\ H)\ H & (c \lhd e, e \lhd^* d)
\end{cases}
\end{array}
$$

$IsRep^c_d$ は見かけの型 c を持つオブジェクトがクラス d のインスタンスであるかどうかを判定する述語であるが，これを判定するには，型 c のオブジェクト参照が指すタプルから祖先クラスの方向にクラス d のサブヒープに格納されるタプルまでリンクしていることを調べればよい．子クラスのタプルとリンクしているかどうかは，子クラスに型変換した結果のオブジェクトがストアに存在するかで判別することができる．この述語は $c = d$ と $c \lhd^+ d$ の 2 つの場合に分けて定義される．$c = d$ である場合，c オブジェクトからどの子クラス e_j のタプルにもリンクしていなければよい．つまり，c オブジェクトを子クラス e_j に型変換した結果のオブジェクトがストアに存在しなければよい．$c \lhd^+ d$ である場合，c オブジェクト参照を d を子孫とする子クラス e に型変換し，それが d のインスタンスであればよい．いずれの場合も c オブジェクトがストアに存在していなければならない．

3.3.5　ストア型の生成

ここからは，ヒープメモリをストアに抽象化し，オブジェクト指向理論を導出していく．

型 $store$ は，型 $Heap$ の部分集合から生成される．HOL において，既存の型 t_1 から新しい型 t_2 を生成するためには次のステップを踏む．

1. t_2 を表現する t_1 の部分集合を定める特徴述語 $p : t_1 \rightarrow bool$ を定義する．
2. この集合が少なくとも一つ要素を持つこと，つまり，$\exists x.\ p\ x$ を証明する．
3. t_1 と，p によって定義される t_2 の部分集合の間に全単射が存在することを表明する．

まず，$Heap$ 型の部分集合を定める特徴述語 $IsStoreRep$ を次のように定義する．

$$
\begin{array}{l}
IsStoreRep\ H \equiv \forall P.\ IsInv\ P \Rightarrow P\ H \\
\quad where \\
\quad\quad IsInv\ P \equiv P\ EmpRep \wedge \\
\quad\quad\quad \bigwedge_c (\forall o\ x\ H.\ P\ H \Rightarrow P\ (SetRep^c\ x\ H)) \wedge \\
\quad\quad\quad \bigwedge_c (\forall H.\ P\ H \Rightarrow P\ (Snd\ (NewRep^c\ H)))
\end{array}
$$

$IsStoreRep$ によって定義される $Heap$ の部分集合は，$Heap$ の要素のうち，次の帰納法により証明される不変表明 P を満たすものの集合である．

- 初期段階：$EmpRep$ が P を満たすことを証明する．
- 帰納段階：P がある $Heap$ の要素について成り立つと仮定し，それに $SetRep^c$ または $NewRep^c$ を適用して得られる要素についても P が成り立つことを証明する（合計 $2|C|$ ステップ）．

言い換えれば，$IsStoreRep$ によって定義される $Heap$ の部分集合は，$EmpRep$，及び，$EmpRep$ に $SetRep^c$ または $NewRep^c$ を任意回適用して得られる要素である．このように定義するのは，属性更新またはオブジェクト生成を通してのみしかストアの状態を変更できないようにするためである．

次に，この部分集合に少なくとも一つ要素が存在することを証明する．これは次の定理として証明される．

$$th \equiv \vdash IsStoreRep\ EmpRep$$

最後に，$store$ と $Heap$ の部分集合の間に全単射が存在することを表明する．これは HOL が提供する ML 関数 $new_type_definition(store, th)$ を呼び出すことにより自動的に表明される．この全単射をそれぞれ次のように定義する．

$$RepStore : store \rightarrow Heap$$
$$AbsStore : Heap \rightarrow store$$

以上の手続きにより，$store$ 型が生成される．図 3.6 はヒープメモリとストアの対応を図示したものである．

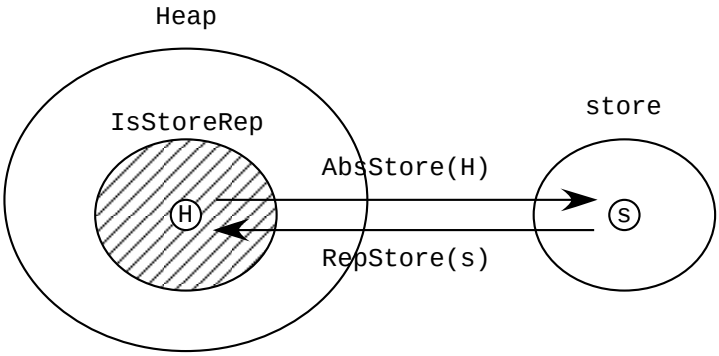

図 3.6: ヒープメモリからストアへの抽象化

3.3.6　演算子の定義

オブジェクト指向理論における定数，演算子は，それらのヒープにおける具体表現と関連付けることにより定義される．これらは全単射 $RepStore$ と $AbsStore$ を用いて次のように定義される．

$$
\begin{aligned}
&Emp \equiv AbsStore\ EmpRep\\
&Ex^c\ o\ s \equiv ExRep^c\ o\ (RepStore\ s)\\
&Get^c\ o\ s \equiv GetRep^c\ o\ (RepStore\ s)\\
&Set^c\ o\ x\ s \equiv AbsStore\ (SetRep^c\ o\ x\ (RepStore\ s))\\
&Cast^c_d\ o\ s \equiv CastRep^c_d\ o\ (RepStore\ s)\\
&Is^c_d\ o\ s \equiv IsRep^c_d\ o\ (RepStore\ s)\\
&New^c\ s \equiv let\ (o, H) = NewRep^c\ (RepStore\ s)\ in\ (o, AbsStore\ H)
\end{aligned}
$$

Get^c_a, Set^c_a は Get^c, Set^c を使って次のように定義される．

$$
Get^c_a\ o\ s \equiv
\begin{cases}
if\ Ex^c\ o\ s\ then\ getElem^i_N\ (Get^c\ o\ s) & \\
\quad else\ Unknown^c_a & (a \in \mathcal{M}_{attr}(c))\\
Get^d_a\ (Cast^c_d\ o\ s)\ s & (d \lhd c, a \in attr(d))
\end{cases}
$$

$$
Set^c_a\ o\ x\ s \equiv
\begin{cases}
if\ Ex^c\ o\ s\ then & \\
\quad let\ t = setElem^i_N\ x\ (Get^c\ o\ s)\ in & \\
\quad\quad Set^c\ o\ t\ s & \\
\quad else\ s & (a \in \mathcal{M}_{attr}(c))\\
Set^d_a\ (Cast^c_d\ o\ s)\ x\ s & (d \lhd c, a \in attr(d))
\end{cases}
$$

Get^c_a は，属性 a がクラス c 自身で定義されたものか，または，祖先のクラスの属性を継承したものか，の 2 つの場合に分けて定義される．a が c で定義されている場合，Get^c により c オブジェクトの属性タプルを取得し，$getElem^i_N$ によりそのタプルの a の要素を取得する．ここで，N は c で定義される属性集合の要素数で，i はその属性集合における a の位置である．c オブジェクトが存在しない場合はその属性の未定義の値を意味する定数 $Unknown^c_a$ を返す．a が祖先クラス d で定義されている場合，c オブジェクトを d に型変換し，Get^c_a を適用する．

Set^c_a も同様に 2 つの場合に分けて定義される．a が c で定義されている場合，Get^c により c オブジェクトの属性タプルを取得し，$setElem^i_N$ によりそのタプルの a の要素を更新する．更新されたタプルを Set^c によりストアに反映する．c オブジェクトが存在しない

場合はストアに変更を加えない．a が祖先クラス d で定義されている場合，c オブジェクトを d に型変換し，Set_a^c を適用する．

3.3.7 公理の導出

オブジェクト指向理論に含まれる公理は以上の定義から導出することができる．各公理の導出過程はパターン化しており，それぞれのパターン応じてカスタマイズされたタクティックにより自動的に証明される．

公理はその導出法により 2 種類に分類される．一つは定義から直接導出できるものであり，もう一つはヒープメモリ上の不変表明として導出できるものである．後者は，公理 3, 7, 14, 15, 16, 17 である．不変表明は $IsInv$ を満たす述語であり，その証明はヒープメモリの構造に基づく帰納法となっている．

不変表明の導出法を公理 6 を例にとって説明する．この公理を次の述語 Inv として定義し，$\vdash \forall s.\ Inv s$ を証明する．

$$Inv\ s \equiv \forall o.\ Ex^c\ o\ s = Is_{d_1}^c\ o\ s \vee ... \vee Is_{d_n}^c\ o\ s \quad (d_i \in \{d \mid c \lhd^* d\})$$

まず，この述語のヒープメモリにおける表現 $InvRep$ を次のように定義する．

$$InvRep\ H \equiv \forall o.\ ExRep^c\ o\ H = IsRep_{d_1}^c\ o\ H \vee ... \vee IsRep_{d_n}^c\ o\ H$$

次に，$InvRep$ がヒープメモリにおける不変表明であること，つまり，

$$\vdash IsInv\ InvRep$$

を証明する．この証明は，$IsInv$ を展開した後，帰納法の以下の各ステップ（合計 $1+2|C|$ 個）を証明すればよい．

$$\begin{aligned}
&\vdash InvRep\ EmpRep \\
&\vdash \forall o\ x\ H.\ InvRep\ H \Rightarrow InvRep\ (SetRep^c\ x\ H)) \quad (c \in C) \\
&\vdash \forall H.\ InvRep\ H \Rightarrow InvRep\ (Snd\ (NewRep^c\ H)) \quad (c \in C)
\end{aligned}$$

次に，この定理と $IsStoreRep$ の定義を用いて，

$$\vdash \forall H.\ IsStoreRep\ H \Rightarrow InvRep\ H$$

を証明する．これは，H がストアを表現する要素ならば，不変表明 $InvRep$ を満たすことを意味する．

また，ストアとヒープメモリの間の全単射を定義した際に自動的に証明される定理から次の定理が導出できる．

$$\vdash \forall s.\ IsStoreRep\ (RepStore\ s)$$

これら 2 つの定理から次の定理を導出する．

$$\vdash \forall s.\ InvRep\ (RepStore\ s)$$

つまり，ストア s のヒープメモリにおける表現は不変表明 $InvRep$ を満たす．

最後に，この定理と演算子 Ex^c, Is^c_d の定義を用いて，求める定理

$$\vdash \forall s.\ Inv\ s$$

が得られる．

その他の不変表明も同様に証明することができる．

第4章　UMLの検証

本章では，ObjectLogicを用いた検証例として，UMLシーケンス図の検証を述べる．シーケンス図は，複数のオブジェクト間の逐次的なメソッド呼び出し系列を矢印やタイムラインで図解したものであり，その系列はHOLの体系において関数適用列として表現しやすい．本章では，シーケンス図により表されるメソッドが，事前事後条件を満たすこと，また，不変表明を満たすことを証明する方法を述べる．例題として簡単な図書館システムの検証を行う．

4.1　図書館システム

ここで証明対象とする図書館システムは，図書館における利用者の登録や本の登録，貸出，返却手続きといった業務を支援するシステムである．

4.1.1　クラス図

クラス図を図4.1に示す（メソッドは貸出手続きに関わるもののみ表示している）．

library クラスはこのシステム全体を管理するクラスであり，貸出や返却など，外界とのインターフェースとなるメソッドを持つ．属性max, daysはそれぞれ，利用者の最大貸出数，最大貸出日数を表す．属性nextcidは利用者の登録の際，次に発行される利用者ID を保持する．同様に, nextiidは物品の登録の際，次に発行される物品IDを保持する．属性customerListは登録されている利用者オブジェクトのリストである．属性itemListは登録されている物品オブジェクトのリストである．属性lendListは貸出オブジェクトのリストである．

customer クラスは図書館に登録される利用者のクラスである．属性id, nameはそれぞれ，利用者ID，名前を表す．属性lendListは貸出オブジェクトのリストである．

item クラスは図書館に登録される物品のクラスである．属性id, titleは物品ID，物品のタイトルを表す．属性lendは貸出オブジェクトを保持する．物品には，本とCDの2種類があり，それぞれitemクラスの子クラスbook, cdとする．bookクラスはISBNを表す属性isbnを持つ．

lend クラスは貸出情報を保持するクラスである．属性customer, itemはそれぞれ貸出を行っている利用者，貸出対象となっている物品である．属性daysは残り貸出日数を表す．この値が負になると貸出が延滞されていることになる．

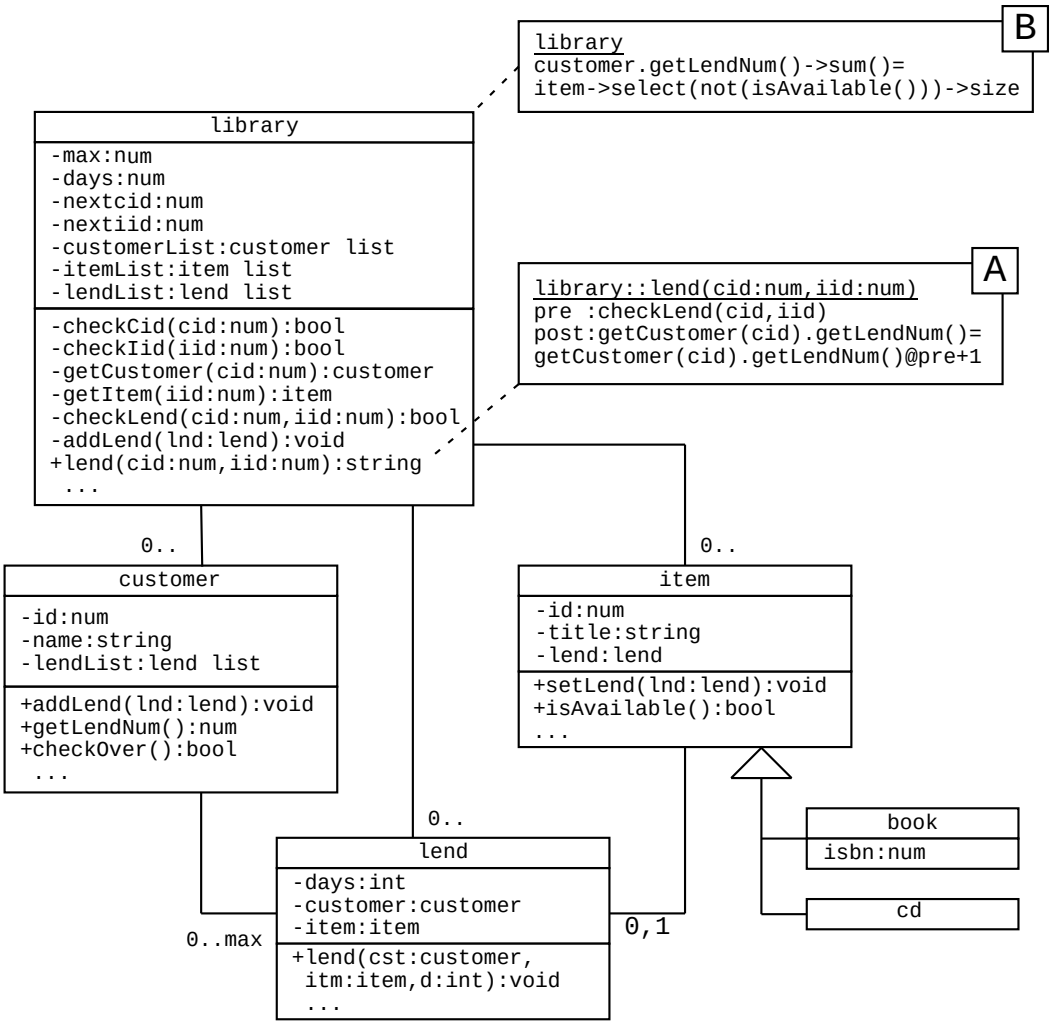

図 4.1: 図書館システムのクラスモデル

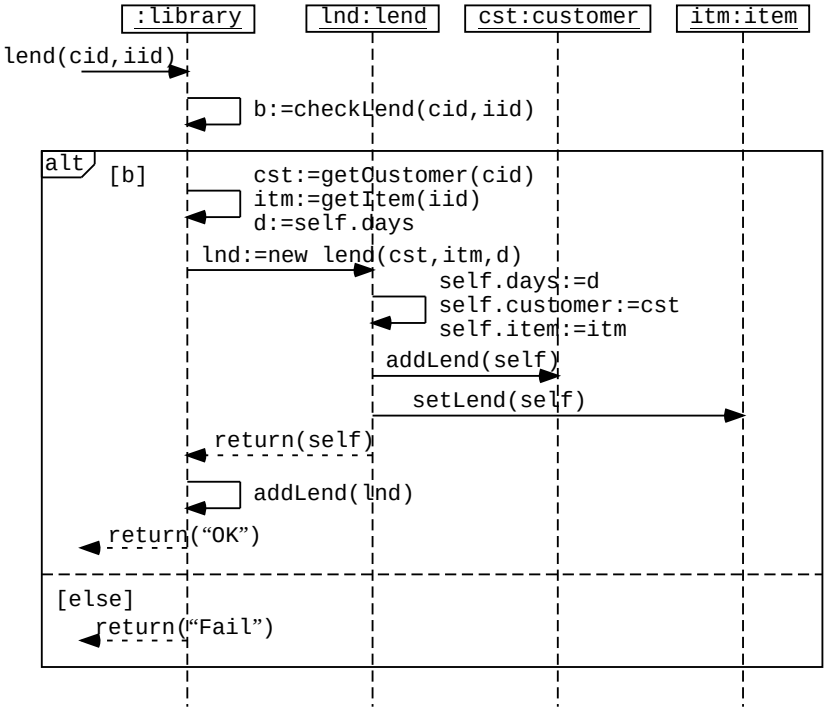

図 4.2: library::lend()

4.1.2 シーケンス図

librar クラスの貸出手続きを行うメソッド lend() と，その内部で呼ばれるメソッド checkLend(), getCustomer() のシーケンス図を示す．

library.lend()

library クラスのメソッド lend() のシーケンス図を図 4.2 に示す．入力は，貸出を行う利用者 ID cid と貸出対象の物品 ID id である．

まず，メソッド checkLend() により cid と iid に対応する利用者と物品がともに貸出可能な状態にあるかチェックする．貸出可能な状態にない場合は，貸出失敗を表すメッセージ"Fail"を出力して終了する．貸出可能ならば，メソッド getCustomer(), getItem() により cid に対応する customer オブジェクト，iid に対応する item オブジェクトをそれぞれ取得し，cst, itm とする．また，最大貸出日数を表す属性 days の値を d とする．次に，lend オブジェクトの生成を行う．コンストラクタの内部では，残り貸出日数を表す属性 days の値を d に設定する．また，属性 customer, item にそれぞれ cst, itm を格納し，リンクを張る．さらに，cst のメソッド addLend()，itm のメソッド addLend() により，cst, itm から lend オブジェクトへのリンクを張る．library オブジェクトは lend オブ

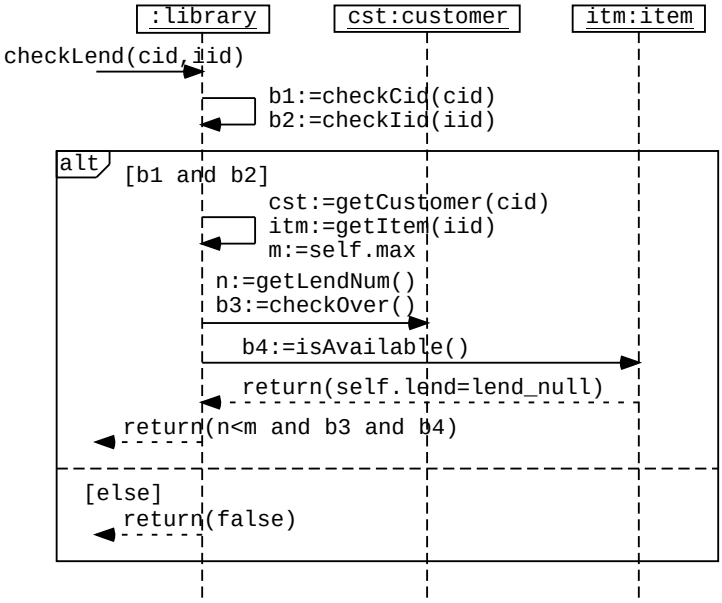

図 4.3: library::checkLend()

ジェクト生成後，メソッド addLend() により生成したオブジェクト lnd を自身にリンクする．最後に，貸出成功を表すメッセージ"OK"を出力し終了する．

library.checkLend()

library クラスのメソッド checkLend() のシーケンス図を図 4.3 に示す．このメソッドは，利用者 ID cid と物品 ID iid を入力し，対応する利用者と物品が貸出可能な状態にあるかどうかをチェックする．チェックする項目は，利用者 ID，物品 ID が有効である（ID に対応する利用者と物品が登録されている）こと，利用者の現在の貸出数が規定される最大貸出数未満であること，利用者が貸出を延滞している物品を保持していないこと，物品が貸出中でないこと，である．

まず，メソッド checkCid(), checkIid() により，cid, iid が有効であるかをチェックする．どちらかが有効でない場合は false を出力し終了する．ともに有効であれば，メソッド getCustomer(), getItem() により，cid, iid に対応する利用者オブジェクト cst，物品オブジェクト itm を取得する．次に，規定の最大貸出数を表す属性 max の値を m とする．また，メソッド getLendNum() により，cst の現在の貸出数を取得し，n とする．n<m であれば貸出数に余裕があることになる．次に，メソッド checkOver() により，cst が貸出を延滞していないかどうかの真偽値を取得し，b3 とする．次に，メソッド isAvailable() により，itm が貸出中でないかどうかの真偽値を取得し，b4 とする．isAvailable() は，lend オブジェクトへのリンクを保持する属性 lend の値が lend_null であるかどうかを

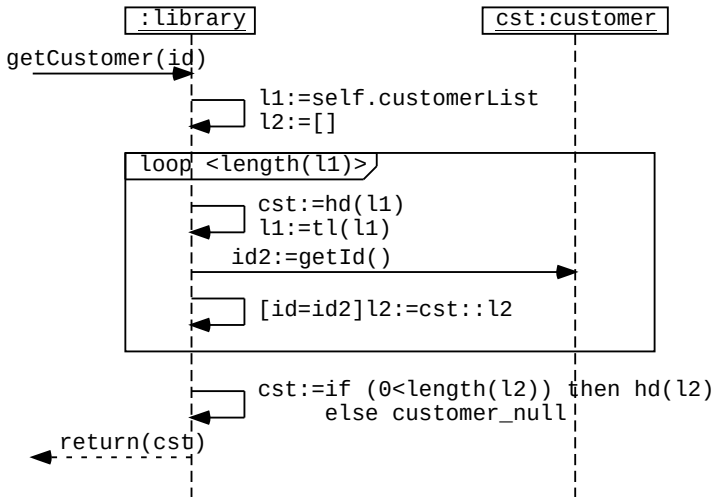

図 4.4: library::getCustomer()

返す述語として定義されている．最後に，b3, b4, n<m の論理積を出力し，終了する．

library.getCustomer()

library クラスのメソッド getCustomer() のシーケンス図を図 4.4 に示す．このメソッドは，入力で与えられる利用者 ID id を持つ利用者オブジェクトを取得するメソッドである．存在しなければ customer_null を返す．

まず，属性として保持する利用者のリスト customerList を l1 とし，l2 を [] とする．次にループを l1 の長さだけ繰り返す．ループ内では，まず，l1 の先頭要素 hd(l1) を cst とし，先頭を除いた残りのリスト tl(l1) を l1 とする．cst について，メソッド getId() により利用者 ID を取得し，id2 とする．id と id2 が一致すれば，cst を l2 に格納し，ループの先頭に戻る．ループ終了後，l2 には id を持つユーザが格納されている（実際にはたかだか 1 つだけ格納される）．最後に，l2 に要素が格納されているならば，その先頭要素を出力し，空であれば，customer_null を出力する．

4.2 HOL における定理証明

以上のように定義される図書館システムに対し定理証明を行う．オブジェクト指向理論においては，

- メソッドの事前事後条件の一貫性
- メソッドが不変表明を満たすこと

の 2 点を証明することが可能である．

まず，メソッドを HOL の関数として定義する．次に 2 つの証明を順に行う．

4.2.1　メソッドの定義

図書館システムのクラスモデルからオブジェクト指向理論を生成し，そのプリミティブ演算子を用いてメソッドを定義する．

図 4.5,4.6 は，library クラスのメソッド lend() の HOL による定義である．シーケンス図から HOL 関数への厳密な変換法は定義していないが，基本的には，逐次的なメソッド実行系列を，ストアに対する関数適用列により表現している．繰り返し構造については，再帰関数により実現する．例えば，getCustomer() の loop 内における，入力 id に対応する利用者オブジェクトを繰り返し取得する部分は，FILTER を用いて実現している．FILTER P l の値は，リスト l の要素のうち述語 P を満たすものだけを取り出したリストである．P を\x. customer_get_id x s = id とすれば，属性 id が入力値 id に一致する利用者オブジェクトだけを取り出すことができる．

4.2.2　メソッド事前事後条件の証明

命題の設定

図書館システムにおいては，貸出手続きが成功すれば，貸出を行った利用者の貸出数が 1 増加する．OCL 制約としては，メソッド lend() に対する事前事後条件として，図 4.1A のように記述することができる．

HOL では次の命題として表される．

```
!lib cid iid s.
  let cst = library_getCustomer lib cid s in
    library_checkLend lib cid iid s /\
    library_ex lib s /\ customer_ex_Inv lib s ==>
      (customer_getLendNum cst (SND (library_lend lib cid iid s)) =
       customer_getLendNum cst s + 1)
```

事前条件は 3 つ存在する．1 つ目は，cid と iid に対応する利用者と物品が貸出可能状態にあるという条件である．残りの 2 つはオブジェクトの存在に関する条件である[1]．1 つは，library オブジェクト lib がストアに存在するという条件である．メソッド library_lend は必ず生成された library オブジェクトに対して起動されるので，この条件を導入することは必然的である．もう 1 つは，lib の属性 customerList に含まれる利用者オブジェクトは必ずストアに存在するという条件であり，次のように定義される．

[1] オブジェクト指向理論に特有の条件であるため OCL 制約としては記述していない．

```
val library_checkCid = Define
  `library_checkCid lib cid s =
     let l = library_get_customerList lib s in
       EXISTS (\x. customer_get_cid x s = cid) l`;

val library_getCustomer = Define
  `library_getCustomer lib cid s =
     let l = library_get_customerList lib s in
       HD (FILTER (\x. customer_get_cid x s = cid) l)`;

val library_checkIid = Define
  `library_checkIid lib iid s =
     let l = library_get_itemList lib s in
       EXISTS (\x. item_get_iid x s = iid) l`;;

val library_get_item = Define
  `library_get_item lib iid s =
     let l = library_get_itemList lib s in
       HD (FILTER (\x. item_get_iid x s = iid) l)`;

val customer_getLendNum = Define
  `customer_getLendNum cst s = LENGTH (customer_get_lendList cst s)`;;

val customer_checkOver = Define
  `customer_checkOver cst s =
   let l = customer_get_lendList cst s in
     EVERY (\x. 0 <= lend_get_days x s) l`;

val item_isAvailable = Define
  `item_isAvailable itm s = (item_get_lend itm s = lend_null)`;

val customer_addLend = Define
  `customer_addLend cst lnd s =
    let l = customer_get_lendList cst s in
      customer_set_lendList cst (lnd::l) s`;
```

図 4.5: HOL におけるメソッド lend() の定義 (1)

```
val library_addLend = Define
  `library_addLend lib lnd s =
    let l = library_get_lendList lib s in
      library_set_lendList lib (lnd::l) s`;

val new_lend = Define
  `new_lend d cst itm s =
    let (lnd,s) = lend_new s in
    let s = lend_set_days lnd d s in
    let s = lend_set_customer lnd cst s in
    let s = lend_set_item lnd itm s in
    let s = customer_addLend cst lnd s in
    let s = item_set_lend itm lnd s in
      (lnd, s)`;

val library_checkLend = Define
  `library_checkLend lib cid iid s =
    if library_checkCid lib cid s /\
      library_checkIid lib iid s then
        let cst = library_getCustomer lib cid s in
        let itm = library_getItem lib iid s in
          customer_getLendNum cst s < library_get_max lib s /\
          customer_checkOver cst s /\ item_isAvailable itm s
    else
      F`;

val library_lend = Define
  `library_lend lib cid iid s =
    if library_checkLend lib cid iid s then
      let cst = library_getCustomer lib cid s in
      let itm = library_getItem lib iid s in
      let d = library_get_days lib s in
      let (lnd, s) = new_lend d cst itm s in
      let s = library_addLend lib lnd s in
        ("OK",s)
    else
      ("Fail",s)`;;
```

図 4.6: HOL におけるメソッド lend() の定義 (2)

```
val customer_ex_Inv = Define
  `customer_ex_Inv lib s =
     library_ex lib s ==>
       !x. MEM x (library_get_customerList lib s) ==> customer_ex x s`
```

これは不変表明であり，システムの起動から常に成立する条件である．つまり，システムの起動後，customerList に customer オブジェクトを追加する際は，必ず customer_new によって生成したものを追加している（利用者登録を行うメソッドをそのように実装している）ため，ストアに存在しないオブジェクトがリストに入り込むことはない．以上の 3 つを前提条件とする．

事後条件は，customer_getLendNum によって取得される値が，library_lend 適用前後のストアで 1 増加するという等式によって表される．

補題の証明 (1)

まず，補題として次の命題を証明する．

```
!lib cid iid s.
  library_checkLend lib cid iid s ==>
    MEM (library_getCustomer lib cid s) (library_get_customerList lib s)
```

この命題は，「貸出条件が成立するならば，貸出を行う利用者は登録されている」を意味する．

まず，library_checkLend の定義を展開する．これにより library_checkCid の項（cid が有効である）が現れる．

```
library_checkCid lib cid s /\ ... ==>
MEM (library_getCustomer lib cid s) (library_get_customerList lib s)
```

次に，library_checkCid, library_getCustomer の定義を展開すると次のゴールが得られる．

```
let l = library_get_customerList lib s in
  EXISTS (\x. customer_get_cid x s = cid) /\ ... ==>
    MEM (HD (FILTER (\x. customer_getCid x s = cid) l)) l
```

これは，次のリストに関する定理の具体化である．

```
|- !P l. EXISTS P l ==> MEM (HD (FILTER P l)) l
```

この定理により書き換えを行えば，証明が完了する．

補題の証明 (2)

もう 1 つの補題として次の命題を証明する．

```
!lib cid iid s. library_ex lib s /\ customer_ex_Inv lib s /\
  library_checkLend lib cid iid s ==>
  customer_ex (library_getCustomer lib cid s) s
```

この命題は,「貸出条件が成立するならば，貸出を行う利用者オブジェクトはストアに存在する」を意味する．

まず，customer_ex_Inv の定義を展開する．

```
library_ex lib s /\  (1)
(library_ex lib s ==>
 !x. MEM x (library_get_customerList lib s) ==> customer_ex x s) /\  (2)
library_checkLend lib cid iid s  (3)
  ==> customer_ex (library_getCustomer lib cid s) s
```

仮定 (1)(2) より，次の新しい仮定が得られる．

```
!x. MEM x (library_get_customerList lib s) ==> customer_ex x s
```

また，先程証明した補題と仮定 (3) により，次の新しい仮定が得られる．

```
MEM (library_getCustomer lib cid s) (library_get_customerList lib s)
```

この 2 つの仮定より，ゴールの結論を導出することができる．

メインの証明

目的の命題の証明を行う．

```
!lib cid iid s.
  let cst = library_getCustomer lib cid s in
    library_checkLend lib cid iid s /\
    library_ex lib s /\ customer_ex_Inv lib s ==>
      (customer_getLendNum cst (SND (library_lend lib cid iid s)) =
       customer_getLendNum cst s + 1)
```

まず，library_lend の定義を展開し，公理により簡単化を行っていくと，次のゴールが得られる．

```
library_checkLend lib cid iid s /\
library_ex lib s /\ library_ex_Inv lib s ==>
```

```
  let cst = library_getCustomer lib cid s in
    LENGTH
      (customer_get_lendList cst
        (customer_set_lendList cst
          (FST (lend_new s)::customer_get_lendList cst s)
          s)) =
    LENGTH (customer_get_lendList cst s) + 1
```

次に，3 つの仮定と補題 (2) より次の仮定が得られる．

```
customer_ex cst s
```

ここでゴールの結論は，同一のオブジェクト `cst` に対する同一の属性 `lendList` の更新と取得となっており，さらに `cst` はストアに存在するオブジェクトであるため，公理 `GET_SET{Get=``customer_get_lendList``}` により書き換え可能である．

```
library_checkLend lib cid iid s /\
library_ex lib s /\ library_ex_Inv lib s ==>
  let cst = library_getCustomer lib cid s in
    LENGTH (FST (lend_new s)::customer_get_lendList cst s) =
    LENGTH (customer_get_lendList cst s) + 1
```

最後に，リストに関する推論規則を適用し，証明が完了する．

4.2.3 不変表明の証明

命題の設定

図書館システムに対する要求の 1 つに,「物品が紛失しないこと」が挙げられる．これは,「全利用者の貸出数の総和は，貸出中の物品の総数に等しい」という不変表明によって表すことができる．OCL 制約としては，`library` クラスに対する不変表明として，図 4.1B のように記述することができる．

HOL では `Inv` として次のように定義される．

```
val Inv = Define
  `Inv lib s = library_ex lib s ==>
     (library_getCustomerLendSum lib s = library_getItemLendSum lib s)`;
val library_getCustomerLendSum = Define
  `library_getCustomerLendSum lib s =
     SUM (MAP (\x. customer_getLendNum x s)
           (library_get_customerList lib s))`;
val library_getItemLendSum = Define
```

```
  `library_getItemLendSum lib s =
     LENGTH (FILTER (\x. ~item_isAvailable x s)
              (library_get_itemList lib s))`;
```

Inv の左辺の library_getCustomerLendSum は全利用者の貸出総数を求める関数である．この関数では，MAP を用いて customer オブジェクトのリストを貸出数のリストに変換する．得られる貸出数のリストに対し，SUM により総和をとることにより貸出数の総和を求めている．一方，Inv の右辺の library_getItemLendSum は貸出中の物品の総数を求める関数である．この関数では，FILTER を用いて item オブジェクトのリストから貸出中である item オブジェクトのリストを取り出す．得られるリストの長さが貸出中の item オブジェクトの個数となる．

この不変表明が library_lend の適用前後で成立することを証明する．証明する命題は次のようになる．

```
!lib cid iid s. Inv lib s /\
  customer_ex_Inv lib s /\ customer_count_Inv lib s /\
  item_ex_Inv lib s /\ item_count_Inv lib s ==>
    Inv lib (SND (library_lend lib cid iid s))
```

つまり，適用前のストアについて，Inv が成立することを仮定し，適用後のストアについて依然として Inv が成立していることを証明する．

Inv 以外の前提条件として，4 つの不変表明が必要となる．customer_ex_Inv は前の証明と同様である．customer_count_Inv, item_ex_Inv, item_count_Inv は次のように定義される．

```
val customer_count_Inv = Define
  `customer_count_Inv lib s =
     library_ex lib s ==>
       let l = library_get_customerList lib s in
         !x. MEM x l ==> (COUNT x l = 1);
val item_ex_Inv = Define
  `item_ex_Inv lib s =
     library_ex lib s ==>
       !x. MEM x (library_get_itemList lib s) ==> item_ex x s;
val item_count_Inv = Define
  `item_count_Inv lib s =
     library_ex lib s ==>
       let l = library_get_itemList lib s in
         !x. MEM x l ==> (COUNT x l = 1);
```

customer_count_Inv は，「library オブジェクト lib の属性 customerList には同一の customer オブジェクトが重複して含まれない」を意味する．COUNT x l はリスト l に含

まれる x の個数である．item_ex_Inv と item_count_Inv は，item オブジェクトに関する customer_ex_Inv, customer_count_Inv と同様の不変表明である．

証明のアウトライン

貸出手続き library_lend を適用する前の全利用者の貸出総数，貸出中の物品の総数をそれぞれ x, y とする．貸出手続き適用前の仮定として「全利用者の貸出総数と貸出中の物品の総数は等しい」という条件が与えられているため，$x = y$ である．

貸出手続き後の全利用者の貸出総数は，貸出条件が成立するか否か（library_checkLend の結果）で次のように変化する．

- (A) 貸出条件が成立するとき，貸出が成功し，全利用者の貸出総数が 1 増加する．
- (B) 貸出条件が成立しないとき，貸出が失敗し，全利用者の貸出総数は変化しない．

同様に，貸出手続き後の貸出中の物品の総数は，次のように変化する．

- (C) 貸出条件が成立するとき，貸出が成功し，貸出中の物品の総数が 1 増加する．
- (D) 貸出条件が成立しないとき，貸出が失敗し，貸出中の物品の総数は変化しない．

これをふまえると，貸出手続き後は，全利用者の貸出総数，貸出中の物品の総数はそれぞれ $x + 1, y + 1$ となる（貸出条件が成立するとき）か，x, y のままである（貸出条件が成立しないとき）かのいずれかである．いずれの場合も，$x = y$ という前提条件より，全利用者の貸出総数，貸出中の物品の総数は等しい値となる．

上の補題 (B)(D) が成立するのは明らかである．これは，貸出条件が成立しないとき，library_lend は何も実行しないからである．

以下では，補題 (A)(C) の証明の概略を示す．

補題 (A) の証明

証明対象の命題は次の通りである．

```
!lib cid iid s.
  library_ex lib s /\  (1)
  customer_ex_Inv lib s /\  (2)
  customer_count_Inv lib s /\  (3)
  library_checkLend lib cid iid s  (4)
  ==>
    (library_getCustomerLendSum lib (SND (library_lend lib cid iid s)) =
     library_getCustomerLendSum lib s + 1)
```

補題 (A) が成立する根拠は次の 3 つの事実である．

- 貸出を行った利用者の貸出数が 1 増加する．
- 貸出を行った利用者とは異なる利用者の貸出数は変化しない．
- 同一の利用者は重複して登録されていない．

最後の事実が必要であるのは，仮に重複して登録されている場合，全利用者の貸出総数をカウントする際に，同じ利用者の貸出数を重複してカウントすることになってしまうからである．

この 3 つの事実から全利用者の貸出総数が 1 増加するという事実を導出することができる．この導出を一般化したものが次の定理である．

```
|- !(x:'a) f g l.
    ((COUNT x l = 1) /\ (f x = g x + 1) /\
        !y. ~(x = y) ==> (f y = g y)) ==>
            (SUM (MAP f l) = SUM (MAP g l) + 1)
```

この定理の意味は次の通りである．いま，あるオブジェクト x がリスト l に 1 つだけ存在し（前提条件の 1 つ目），この x に関数 f を適用して得られる値が，関数 g を適用して得られる値より 1 だけ大きい（前提条件の 2 つ目）とする．また，x とは異なるすべてのオブジェクトに対しては，f を適用して得られる値と g を適用して得られる値は等しいとする（前提条件の 3 つ目）．このとき，l に含まれるすべてのオブジェクトに対して f を適用して得られる値の総和は g を適用して得られる値の総和より 1 だけ大きい．図 4.7 はこの定理を図示したものである．

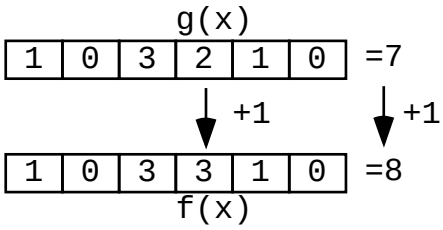

図 4.7: 利用者の貸出総数の変化

この定理の x, f, g, l をそれぞれ

```
library_get_customer lib cid s
\x. customer_get_lendnum x (SND (library_lend lib cid iid s))
\x. customer_get_lendnum x s
library_get_customerlist lib s
```

で特化すれば，次の定理が得られる．

```
(COUNT (library_getCustomer lib cid s)
  (library_get_customerList lib s) = 1) /\  (X)
(customer_getLendNum (library_getCustomer lib cid s)
  (SND (library_lend lib cid iid s)) =
 customer_getLendNum (library_getCustomer lib cid s) + 1) /\  (Y)
(!y. ~(library_getCustomer lib cid s = y) ==>
  (customer_getLendNum y (SND (library_lend lib cid iid s)) =
   customer_getLendNum y s))  (Z)
==>
  (SUM
    (MAP
      (\x. customer_getLendNum x (SND (library_lend lib cid iid s)))
      (library_get_customerList lib s)) =
   SUM
    (MAP
      (\x. customer_getLendNum x s)
      (library_get_customerList lib s)) + 1
```

この定理の結論部は証明対象の命題の結論部と一致する（library_getCustomerLendSum の定義を展開したもの）．したがって，目的の命題を証明するためには，この定理の 3 つの前提条件 (X)(Y)(Z) を，目的の命題の 4 つの前提条件 (1)〜(4) から導出できればよい．

前提条件 (X) は，前提条件 (1)(3)(4) から導出することができる．前提条件 (Y) は，4.2.2 で証明した定理と前提条件 (1)(2)(4) から導出することができる．前提条件 (Z) は，そのまま定理として証明することができる（証明略）．

以上により補題 (A) が証明される．

補題 (C) の証明

証明の流れは補題 (A) と同様である．

証明対象の命題は次の通りである．

```
!lib cid iid s.
  library_ex lib s /\  (1)
  item_ex_Inv lib s /\  (2)
  item_count_Inv lib s /\  (3)
  library_checkLend lib cid iid s  (4)
  ==>
    (library_getItemLendSum lib (SND (library_lend lib cid iid s)) =
     library_getItemLendSum lib s + 1)
```

補題 (C) が成立する根拠は次の 3 つの事実である．

- 貸出対象となった物品が貸出中となる．
- 貸出対象となった物品とは異なる物品の貸出状態は変化しない．
- 同一の物品は重複して登録されていない．

この 3 つの事実から貸出中の物品の総数が 1 増加するという事実を導出することができる．この導出を一般化したものが次の定理である．

```
|- !(x:'a) f g l.
    ((COUNT x l = 1) /\ f x /\ ~(g x) /\
        !y. ~(x = y) ==> (f y = g y)) ==>
            (LENGTH (FILTER f l) = LENGTH (FILTER g l) + 1)
```

この定理の意味は次の通りである．いま，あるオブジェクト x がリスト l に 1 つだけ存在し（前提条件の 1 つ目），この x に関数 f を適用して得られる値が真であり（前提条件の 2 つ目），関数 g を適用して得られる値が偽である（前提条件の 3 つ目）とする．また，x とは異なるすべてのオブジェクトに対しては，f を適用して得られる値と g を適用して得られる値は等しいとする（前提条件の 4 つ目）．このとき，l に含まれるすべてのオブジェクトに対して f を適用して真となるオブジェクトの数は，g を適用して真となるオブジェクトの数より 1 だけ大きい．図 4.8 はこの定理を図示したものである．

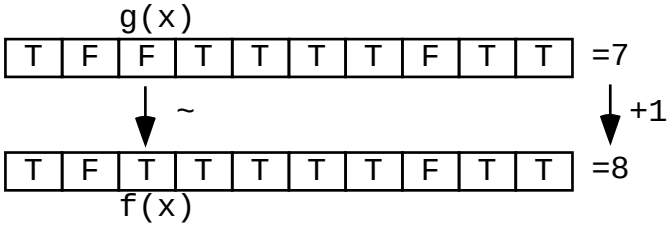

図 4.8: 貸出中の物品総数の変化

この定理の x, f, g, l をそれぞれ

```
library_get_item lib iid s
\x. ~item_is_available x (SND (library_lend lib cid iid s))
\x. ~item_is_available x s
library_get_itemlist lib s
```

で特化すれば，次の定理が得られる．

```
(COUNT (library_getItem lib iid s)
  (library_get_itemList lib s) = 1) /\  (S)
~item_isAvailable (library_getItem lib iid s)
```

```
  (SND (library_lend lib cid iid s))  /\  (T)
item_isAvailable (library_getItem lib iid s) s /\  (U)
(!y. ~(library_getItem lib iid s = y) ==>
  (~item_isAvailable y (SND (library_lend lib cid iid s)) =
   ~item_isAvailable y s))  (V)
==>
  (LENGTH
    (FILTER
      (\x. ~item_isAvailable x (SND (library_lend lib cid iid s)))
      (library_get_customerList lib s)) =
   LENGTH
    (FILTER
      (\x. ~item_isAvailable x s)
      (library_get_customerList lib s)) + 1
```

この定理の結論部は証明対象の命題の結論部と一致する（library_getItemLendSum の定義を展開したもの）．したがって，目的の命題を証明するためには，この定理の 4 つの前提条件 (S)(T)(U)(V) を，目的の命題の 4 つの前提条件 (1)〜(4) から導出できればよい．

前提条件 (S) は，前提条件 (1)(3)(4) から導出することができる．前提条件 (T) は，「貸出対象となった物品は貸出不能になる」を意味する定理

```
|- !lib cid iid s.
   library_checkLend lib cid iid s /\
   library_ex lib s /\ item_ex_Inv lib s ==>
     ~item_isAvailable (library_getItem lib iid s)
       (SND (library_lend lib cid iid s))
```

と（証明略），前提条件 (1)(2)(4) から導出することができる．前提条件 (U) は，前提条件 (4) から導出することができる．前提条件 (V) は，そのまま定理として証明することができる（証明略）．

以上により，補題 (B) が証明される．

第5章　関連研究

5.1　Deep embedding vs shallow embedding

形式言語を定理証明器に実装する手法には，deep embedding と shallow embedding の 2 通りの手法が存在する [6]．Deep embedding は，言語のシンタックスを型として定義し，その型の要素と理論の要素のマッピング関数を定義することにより意味論を定義する手法である．例えば，論理演算は次のように定義される．

```
Hol_datatype ‘exp = TRUE | AND of exp => exp | NOT of exp‘;
Define ‘(sem TRUE = T) /\
        (sem (AND exp exp) = sem exp /\ sem exp) /\
        (sem (NOT exp) = ~sem exp)‘;
```

一方，shallow embdding は，言語の要素を，直接，理論の要素によって表現することにより意味論を定義する手法である．この場合，論理演算は次のように定義される．

```
Define ‘TRUE = T‘;
Define ‘AND exp exp = exp /\ exp‘;
Define ‘NOT exp = ~exp‘;
```

Deep embedding は言語のメタレベルの性質（例えば，言語の型システムが安全である，などの）の証明に適しており，shallow embedding は言語の個々のインスタンスの性質（例えば，変数 x の値が 10 以下である，など）の証明に適している．

我々の目的は，アプリケーションの検証であるため，shallow embedding を採用している．例えば，属性は get 演算子と set 演算子で表現し，継承は cast 演算子や is 演算子で表現している．Shallow embedding がアプリケーションレベルの証明に適する理由は，deep embedding に比べて，証明ステップ数が少なくなるという点である．Deep embedding においてアプリケーションレベルの証明をしようとした場合，意味関数 `sem` を介している分だけ証明ステップが多くなってしまう．Shallow embedding のもう 1 つの利点は，アプリケーションレベルの型検査を受けることができる点である．例えば，deep embedding においてオブジェクトの型を定義する場合，オブジェクトの種類は複数あるにもかかわらず，`OBJ` といったメタレベルの統一的な型で定義されてしまう．しかし，shallow embedding では，`fig` や `rect` といったアプリケーションレベルの別々の型を割り当てることができる．これは，ML や HOL においてメソッドを定義していく際に，特に有効である．例えば，`fig` オブジェクトに対し，誤って `rect` クラスのメソッドを呼ぶように定義した場合，

型エラーとして検出することができる．これにより，安全なモデルを効率よく構築することが可能となる．

5.2　ヒープメモリに基づくオブジェクト指向理論

Berg ら (LOOP[8]) は，JML 仕様の付加された Java プログラムの検証の意味論として，ヒープメモリ構造を Isabelle/HOL と PVS に一般的な理論として定義している．ヒープメモリの構成要素は型無しのブロックであり，任意の Java オブジェクトを格納することができる．型無しブロックは簡単に表現すれば，次のようなリストと直積を使った型として定義される．

```
int list # bool list # ref list
```

ここで，Java に出現する型は `int`, `bool`, `ref` の 3 つであるとする．`ref` はオブジェクト参照の型である．このブロックには一つのオブジェクトのすべての属性を格納することができる．例えば，`int` 型の属性 `x=2`, `y=3`，`bool` 型の属性 `b=T` を持つオブジェクトは

```
([2,3],[T],[])
```

のようにブロックに格納される．どの属性がどの位置に格納されるかという情報は別に与えられる．このような型無しブロックを定義すれば，任意の Java オブジェクトを格納できるヒープメモリを一般的に構築することができる．

このような定義が可能であるのは，ヒープメモリに格納するデータ型が，Java に出現する有限個に限られるからである．我々は，プログラミング言語ではなく分析モデルを検証対象としているため，対象システムに出現する任意の型を扱える必要がある．任意の型を格納するためのメモリ構造を一般的な理論として簡潔な形で定義することは HOL の型制約では困難であるため，対象システムのクラスモデルからそれに特化した理論を自動生成するという手段を選んだ．

勿論，LOOP の理論においても，Java クラスとして任意のデータ型を表現することは可能である．しかし，それらのクラスから型の性質を導出するための証明が余分に必要となる．本理論の長所は，HOL のライブラリとして既にその性質が証明された多種多様な型を直接オブジェクトの属性に組み込んで検証することができることである．また，新しい型が必要な場合も，その型の構築は HOL のプロセスとしてオブジェクトに関する推論とは独立して行うことができる．

5.3　Extensible record による構造的サブタイピング

Naraschewski ら [11] は，Isabelle/HOL において，オブジェクトを extensible record と呼ばれるデータ型により表現している．Extensible record は

```
{x=3,y=5,...}
```

のように表記されるデータ型であり,「...」の部分に新たな要素を追加することにより,

```
{x=3,y=5,f=true,...}
```

と拡張することができる．このデータ型は，最後の要素を型変数としたタプル `int#int#'a` で表現されており，`'a` を `bool#'a` で詳細化することにより拡張を行っている．継承は，サブクラスにおいてレコードの要素を追加していくことにより実現される．

このようなデータ型によりオブジェクトの構造的なサブタイピングが可能となり，オーバーライドや動的束縛といった概念を実現することができる．しかし，オブジェクトが参照としての役割を持たないため，協調動作を実現することは不可能である．協調動作が実現できることは，それをベースに機能が構成されるアプリケーションの検証にとって特に重要である．したがって本理論では，ストアという概念を導入し，オブジェクトをそれに格納されるデータへの参照として表現することにより，協調動作を可能とした．

5.4 Z との比較

Z[15] のような仕様記述言語と ObjectLogic の違いは，オペレーションの定義方法にある．Z では，オペレーションの定義は，事前事後条件を付加することによって行う．一方，ObjectLogic では，オペレーションの定義は，その内部動作を ML や HOL における具体的な関数として記述することによって行う．事前事後条件は関数定義から導出されるべきものである．

ObjectLogic がこのようなスタイルをとっているのは，モデルを実行しながら構築していくことを可能とするためである．ML では，インタプリタと対話しながら関数を定義し，即座に実行結果を確認することができる．このため，Z のような宣言的なスタイルに比べ，オペレーションの動作を直感的に理解しやすいという利点がある．

第II部

ファイアウォールサーバのモデル化と検証

第6章　ファイアウォールサーバの概要

本書ではファイアウォールサーバの主要機能であるパケットフィルタリングシステムについて，モデル化と検証を行う．本章ではまず，パケットフィルタリングシステムの仕様を述べ，次に，具体的なパケット処理例を示す．

6.1　パケットフィルタリングシステムの仕様

対象とするファイアウォールは，ネットワーク間を通過するパケットを接続状態，フィルタルールに基づき，通過許可，拒否の判定を行うステートフルパケットフィルタである．また，パケットのアドレス変換，DoS 攻撃の検知を行う．以下，その仕様を示す．

1. ファイアウォールは，内部ネットワークと外部ネットワークの 2 つのネットワークをつなぐものとする．
2. アウトバウンド送信（内部から外部への送信）においては，フィルタルールを満たすパケットのみ通過を許可する．
3. インバウンド送信（外部から内部への送信）においては，アウトバウンド送信に対する返信パケットのみ通過を許可する．
4. 同一接続の 2 回目以降のパケット送受信については無条件で通過を許可する．
5. 接続数には上限があり，いっぱいであればパケットを拒否する．
6. 接続の最大数はユーザが設定可能である．
7. 接続はそれに属す最後のパケットが通過後，一定時間経過したとき自動的に切断する．
8. フィルタルールは以下の 5 つの TCP/IP ヘッダ情報をもとにパケットの通過許可，拒否の定義を行う．
   - 送信元 IP アドレス
   - 送信元ポート番号
   - 宛先 IP アドレス
   - 宛先ポート番号

- プロトコル（TCP，UDP）

9. フィルタルールはユーザが設定可能である．
10. フィルタルールが定義されていない場合は，パケットをすべて拒否する．
11. アドレス変換は PAT(Port Address Translation) 方式で行う．つまり，内部ホストのプライベート IP アドレスとポート番号を，ファイアウォールが持つ公開 IP アドレスとポート番号にマップする．
12. アドレス変換規則の割り当ては動的に行う．変換規則は最後に使用されてから一定時間経過後に自動的に消去する．
13. アドレス変換規則のタイムアウト時間はユーザが設定可能である．
14. 公開 IP アドレスは 1 つだけとする．
15. 公開ポート番号はユーザが設定可能である（複数可）．
16. ポートに空きがない場合はパケットを拒否する．
17. アウトバウンド送信においては，送信元アドレスの変換を行い，インバウンド送信においては宛先アドレスの変換を行う．
18. アウトバウンド送信，インバウンド送信ともに，秒間に拒否されたパケット数が規定の閾値を超えた場合，DoS 攻撃の可能性とみなし，警告を発生させる．
19. DoS カウンタの閾値はユーザが設定可能である．
20. パケットフィルタ機能はユーザが起動，及び，停止を行う．
21. パケットフィルタ機能が停止中はパケットはすべて拒否する．
22. パケットフィルタ機能が停止時には，すべての接続が切断される．
23. パケットフィルタリングに関するコンフィギュレーションの変更（公開 IP アドレス，公開ポート番号，フィルタルール，接続タイムアウト時間，アドレス変換規則タイムアウト時間，DoS カウンタ閾値）は，パケットフィルタ機能が停止中のみ可能である．

（注意点）このファイアウォールが管理する接続は，TCP 接続とは別のものである．TCP では接続を確立するまでに 3 ウェイハンドシェイクを行うが，このファイアウォールは，SYN や ACK のフラグ値までは参照せず，最初のアウトバウンドパケットが送出された時点で接続を生成する．接続を切断する際も，FIN フラグを参照するといったことはせず，タイムアウトにより自動的に切断する．したがって，ファイアウォールに接続が生成されたとしても，両端のホストの間に TCP 接続が確立しているとは限らないし，TCP 接続が

確立していたとしてもファイアウォールに接続が存在するとは限らない．ここでの接続は，通信が許可された 2 つのアドレスを記録するために用いられるものである．

6.2 パケットフィルタリングの例

対象とするファイアウォールのパケットフィルタリング方式をアウトバウンド，インバウンド両方向のトラフィックについて例を用いて説明する．アドレスは 10.0.0.1 のような形式ではなく，整数値で抽象化する．0...99 をプライベートアドレスの予約範囲とし，それ以上をグローバルアドレスとする．以降ではアドレスといった場合，IP アドレスとポートの組を意味し，(200,1000) のように表す．ファイアウォールの公開 IP アドレスは 200 とし，3 つのポート 1200, 1210, 1220 を持つとする．最大接続数は 3 とする．フィルタールールは以下のように指定されているとする．

- 送信元アドレス 10, 20, 30 を許可
- 送信元ポート 1050, 1070, 1100 を許可
- 宛先アドレス 250 のみ許可
- 宛先ポート 80 のみ許可
- プロトコルは TCP のみ許可

6.2.1 アウトバウンド送信の例

いま，図 6.1 に示すように，IP アドレス 30 の内部ホストが，IP アドレス 250 の外部ホストに送信元アドレス (30,1100)，宛先アドレス (250,80) の TCP パケットを送信しようとしているとする．パケット内の値 `src`, `dst`, `port` はそれぞれ，送信元アドレス，宛先アドレス，プロトコルを表す．既に 2 つの接続が張られており，接続表に記録されている．接続表の最初に記録されている接続は，アドレス (20,1070) の内部ホストとアドレス (250,80) の外部ホストの間に張られたものである．変換表には内部アドレスを公開アドレスにマッピングするための規則が定義されている．最初の規則は，内部アドレス (20,1070) を公開アドレス (200,1200) に変換することを定義している．IP アドレス 200 はファイアウォールが持つただ 1 つの公開 IP アドレスである．公開ポート 1200 はファイアウォールによって動的に与えられた値である．2 番目の接続についても同様である．

アウトバウンド送信の流れを以下に示す．

1. パケットフィルタが作動中であるかどうかのチェック
2. 既存の接続に属すパケットであるかのチェック
3. 接続容量，ポートに空きがあるかのチェック

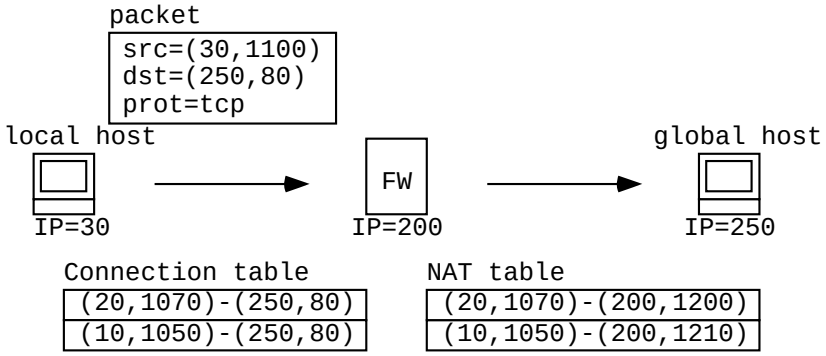

図 6.1: アウトバウンドフィルタリングの例 (1)

4. フィルタルールの条件に合致するかのチェック
5. 接続の追加
6. 送信元アドレス変換

まず，パケットフィルタが作動中であるかをチェックする．停止中であればすべてのパケットの通過を拒否する．次に，パケットが既存の接続に属すものであるかをチェックする．このパケットの送信元アドレス，宛先アドレスの組 (30,1100)-(250,80) は接続表に記録されていないので，これが新しい接続であるとわかる．仮にこれが既存の接続のパケット，例えば (10,1050)-(250,80) であれば即座に通過が許可される．次に，接続容量，ポートに空きがあるかをチェックする．接続数，ポート数ともに有限であるから，これらに空きがなければ物理的に送信不可能となる．この場合，まだ接続は 2 つしか張られておらず，まだ空きがある．空きがなければパケットの通過を拒否する．また，ポートについてもまだ 1200 と 1210 の 2 つしか使用されておらず，まだ空きがある．次に，パケットのヘッダ情報がフィルタールールを満たすかチェックする．このパケットは条件を満たすので通過が許可される．通過が許可された時点で新しい接続が確立するので，接続表に新しい接続 (30,1100)-(250,80) を追加する．最後に送信元アドレスの変換を行う．内部アドレス (30,1100) に対応する変換規則はまだ定義されていないので，新しい変換規則 (30,1100)-(200,1220) が自動的に追加される．ポート 1220 は未使用のポートの中から動的に割り当てられたものである．この規則を用いて，パケットの送信元アドレス (30,1100) を (200,1220) に書き換え，外部ネットワークに送出される．図 6.2 は，図 6.1 の状態におけるパケット処理後の状態を示している．

6.2.2 インバウンド送信の例

インバウンド送信については，アウトバウンド送信に対するリプライのパケットのみを許可する．インバウンド送信の流れを以下に示す．

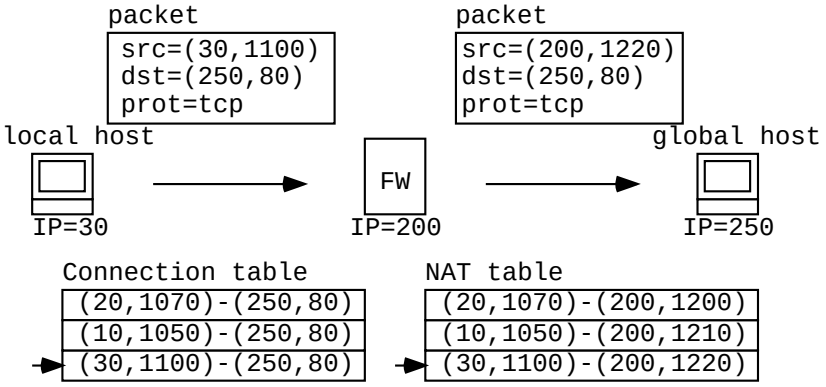

図 6.2: アウトバウンドフィルタリングの例 (2)

1. パケットフィルタが作動中であるかどうかのチェック
2. 宛先アドレス変換
3. 既存の接続のパケットであるかのチェック

図 6.3 は，図 6.2 の状態における，外部アドレス (250,80) からファイアウォールの公開アドレス (200,1220) に向けて送信されたパケットの処理を示している．まず，パケットフィルタが作動中であるかをチェックする．停止中であればパケットの通過を拒否する．次に，パケットの宛先アドレス変換を行う．宛先アドレス変換では，送信元アドレス変換とは逆の向きに変換表を用いる．つまり，右側に記録される公開アドレスから，左側に記録される内部アドレスに変換する．この場合，変換表の 3 番目の規則に基づき，公開アドレス (200,1220) が内部アドレス (30,1100) に変換される．ここで，変換規則が定義されていない場合，例えば宛先アドレスが (200,1230) の場合は，外部から新しく接続を試みようとするパケットであるとみなし，パケットを拒否する．既存の接続のパケットであれば，それ以前に同じ 2 つのアドレス間のアウトバウンド送信が行われ，変換規則が定義されているはずである．ただし，宛先アドレス変換規則が存在していたとしても，接続が存在するとは限らない．例えば，送信元アドレス (300,80)，宛先アドレス (200,1220) のパケットは，宛先アドレス変換規則は変換表の 3 番目に記録されているが，接続表には，アドレス (30,1100) と (300,80) の間の接続は記録されていない．したがって最後に，このパケットが既存の接続のものであるかをチェックする．接続表の 3 番目の値から，アドレス (30,1100) と (250,80) の間に接続が張られていることがわかるので，このパケットは既存の接続のものであるとみなし通過を許可する．接続が記録されていない場合はパケットを拒否する．

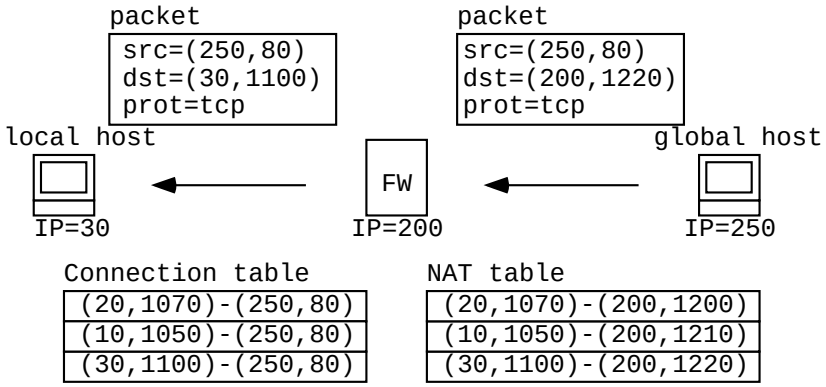

図 6.3: **インバウンドフィルタリングの例**

第7章　UMLによるモデル化

本章では，前章で与えられた仕様に基づき，パケットフィルタリングシステムの UML モデルを定義する．まず，7.1 においてクラス図を示す．次に，7.2 においてシーケンス図を示す．

7.1　クラス図

パケットフィルタリングシステムのクラス図を図 7.1 に示す．pfm(packet filtering manager) はパケットフィルタリング機能全体を管理するクラスであり，4 つのクラスを集約する．contable は接続表を表すクラスであり．接続の追加や更新を行う．接続表に格納される個々の接続は connection クラスにより表す．nattable はアドレス変換表を表すクラスであり，アドレス変換や変換規則の動的生成などを行う．変換表に格納される個々の変換規則は natrule クラスにより表す．frule はフィルタルールを定義するクラスであり，ルールに基づくパケットの通過許可判定を行う．doscounter は DoS カウンタを表すクラスであり，一定時間に拒否されたパケットをカウントし，DoS 警告を発生させる．packet は処理対象となるパケットを表すクラスである．

pfm クラスは 5 つの属性を持つ．active は，パケットフィルタが作動中であるかどうかの真偽値を表すフラグである．contable, nattable, frule, doscounter は，集約する 4 つのオブジェクトを格納する属性である．メソッドとして，パケットフィルタ機能の起動と停止を行うメソッド start(), stop() や，アウトバウンドパケット，インバウンドパケットを処理するメソッド filterOut(), filterIn()，時間を進めるメソッド incSec() などを持つ．

frule クラスは 5 つの属性を持つ．srcAddrTable は，通過許可する送信元 IP アドレスのリストである．同様に，dstAddrTable, srcPortTable, dstPortTable, protocolTable はそれぞれ，通過許可する宛先アドレス，送信元ポート番号，宛先ポート番号，プロトコルのリストである．メソッドとして，パケットがフィルタルールを満たすかをチェックするメソッド check() などを持つ．

doscounter クラスは 3 つの属性を持つ．alert は，DoS 警告が発せられたときに真となるフラグである．count は，拒否されたパケット数を保持するカウンタである．threshould は，カウンタの閾値であり，これに達すると DoS 警告が発せられる．メソッドとして，カウンタをインクリメントするメソッド inc() などを持つ．

contable クラスは 3 つの属性を持つ．maxSize は，最大接続数である．timeLimit は，

接続のタイムアウト時間である．connectionList は，接続オブジェクトのリストである．メソッドとして，新しい接続を追加するメソッド addConnection() や，接続を更新（タイムアウト時間を初期化）するメソッド update() を持つ．

connection クラスは 3 つの属性を持つ．localAP, globalAP はそれぞれ，接続が張られている内部ホストと外部ホストのアドレス（IP アドレスとポート番号の組）である．timer はタイムアウト残り時間である．

nattable クラスは 4 つの属性を持つ．ipAddr は，ファイアウォールの公開 IP アドレスである．ports は，ファイアウォールの公開ポート番号のリストである．リストの要素は，ポート番号とその使用状態を表す真偽値の組である．timeLimit は，アドレス変換規則のタイムアウト時間である．natruleList は，変換規則オブジェクトのリストである．メソッドとして，送信元アドレス変換，宛先アドレス変換を行うメソッド srcnat(), dstnat() などを持つ．

natrule クラスは 3 つの属性を持つ．localAP は内部アドレスであり，globalAP はその内部アドレスに割り当てられたファイアウォールの公開アドレスである．timer はタイムアウト残り時間である．

packet クラスは 5 つの属性 srcAddr, dstAddr, srcPort, dstAddr, protocol を持つ．それぞれ，送信元 IP アドレス, 宛先 IP アドレス，送信元ポート番号, 宛先ポート番号，プロトコルである．

各クラスのメソッドを図 7.2, 7.3, 7.4 に示す．図 7.2 は，pfm クラスのメソッド filterOut(), filterIn() に関わるメソッドである．図 7.3 は，pfm クラスのメソッド incSec() に関わるメソッドである．図 7.4 は，コンフィギュレーションの設定に関わるメソッドである．

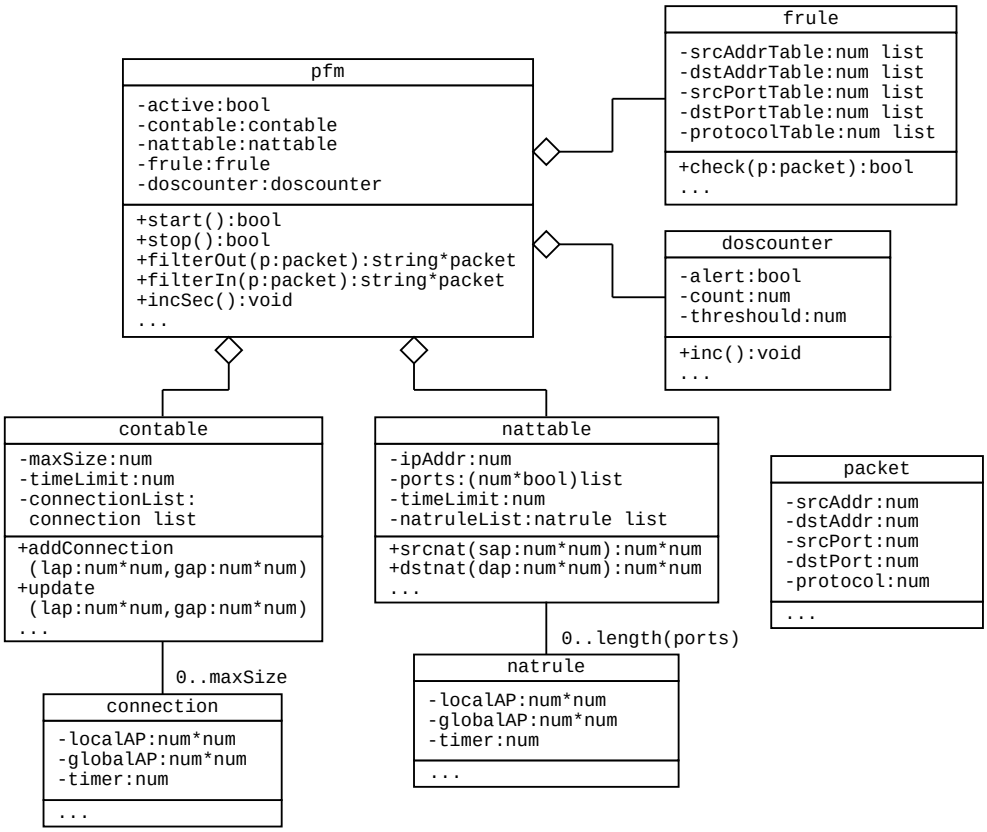

図 7.1: パケットフィルタリングシステムのクラス図

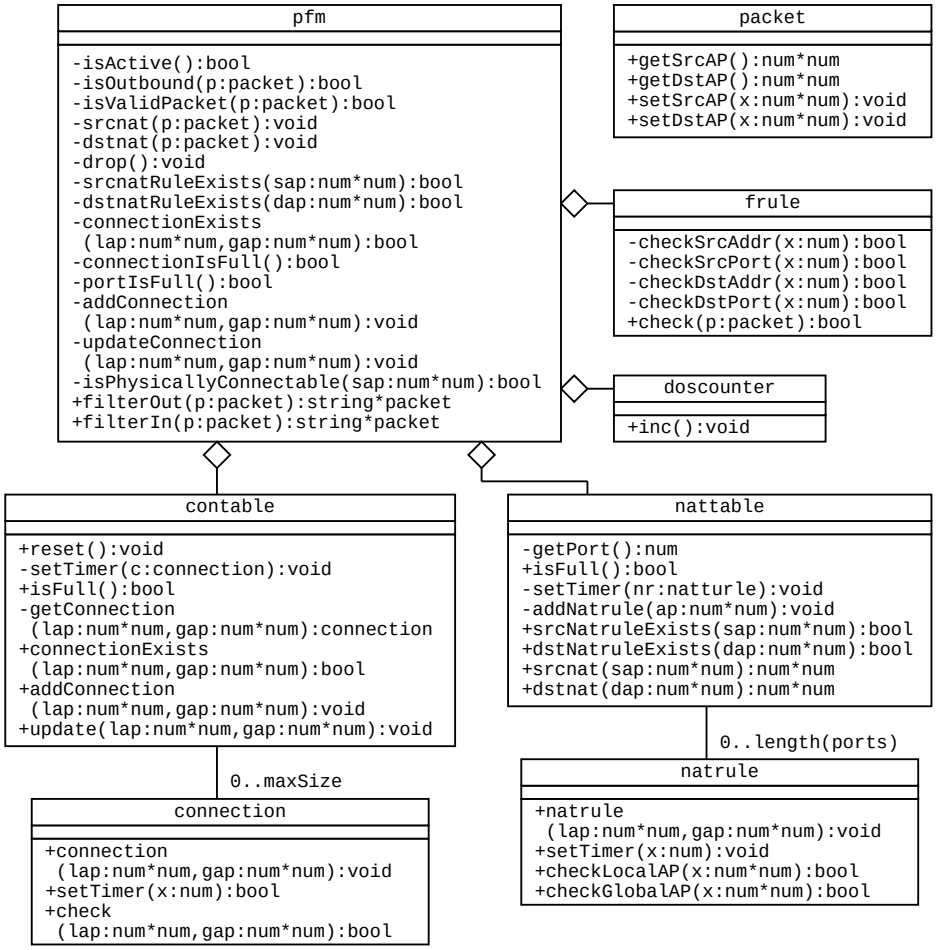

図 7.2: filterOut(), filterIn() **に関わるクラス図**

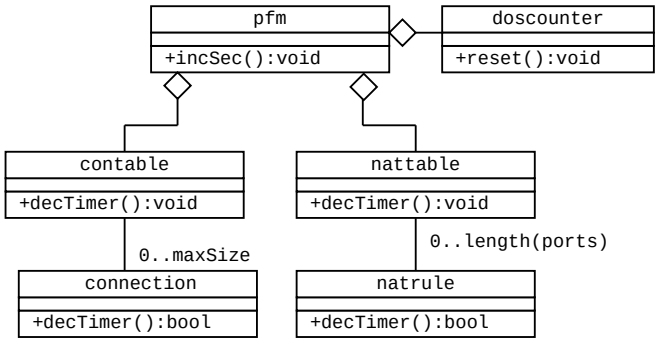

図 7.3: incSec() **に関わるクラス図**

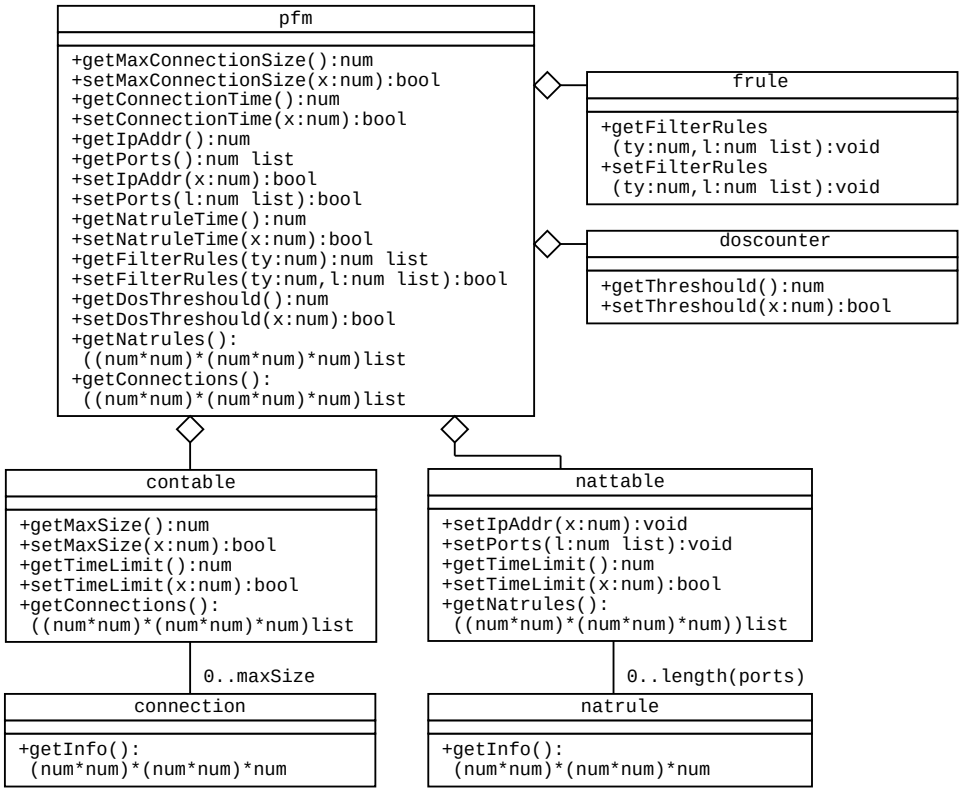

図 7.4: コンフィギュレーションの設定に関わるクラス図

7.2　シーケンス図

7.2.1　pfm.filterOut()

pfm クラスのメソッド filterOut() のシーケンス図を図 7.5 に示す．このメソッドは入力パケット p に対し，アウトバウンドフィルタを適用する．

まず，メソッド isActive() により，パケットフィルタが作動中であるかどうかをチェックする．停止中ならば，パケットを拒否する．パケット拒否時は，出力パケットを packet_null とする．作動中ならば，パケットフィルタ処理を以下のように行っていく．

まず，パケット p のメソッド getSrcAP(), getDstAP() により，送信元アドレス sap，宛先アドレス dap をそれぞれ取得する．次にメソッド connectionExists() により，sap と dap の間に接続が存在するかどうかをチェックする（7.2.3 参照）．接続が存在するとき，このパケットは無条件に通過許可される．このときに行う処理は，接続の更新と送信元アドレス変換である．接続の更新はメソッド updateConnection() により行う．このメソッドの内部では，2 つのアドレス sap, dap 間の接続のタイムアウト時間が初期化される（7.2.9 参照）．送信元アドレス変換は，メソッド srcnat() により行う．このメソッドの内部では，パケット p の送信元アドレスが，変換規則に従い書き換えられる（7.2.6 参照）．これらの処理後，パケット p を出力する．接続が存在しないときは，パケット p は新しい接続を張ろうとするパケットであるから，引き続き通過許可判定を行っていく．

通過許可判定では，接続容量チェック（新しい接続を張るための物理的な容量があるかどうか）とフィルタルール条件チェックの 2 つを行う．まず，メソッド isPhysicallyConnectable() により sap を接続するための物理的容量があるかどうかをチェックする．(7.2.4 参照)．容量がないときはパケットを拒否する．パケット拒否時には，メソッド drop() を呼び出す．このメソッド内部では，リンクする doscounter オブジェクトのメソッド inc() が呼び出され，拒否したパケットの個数が加算される．容量があるとき，フィルタルール条件チェックへと進む．

フィルタルール条件チェックはメソッド isValidConnection() により行う．このメソッドの内部では，パケット p がアウトバウンドパケットであるかをチェックし，さらに，パケットの 5 つのヘッダ情報がフィルタルールに定義される条件を満たすかどうかをチェックする（7.2.11 参照）．条件チェックを満たすとき，通過が許可される．通過許可後は，接続の生成と送信元アドレス変換をそれぞれ，メソッド addConnection()（7.2.10 参照），srcnat() により行い，パケット p を出力する．条件チェックに合わないときは，パケットを拒否する．

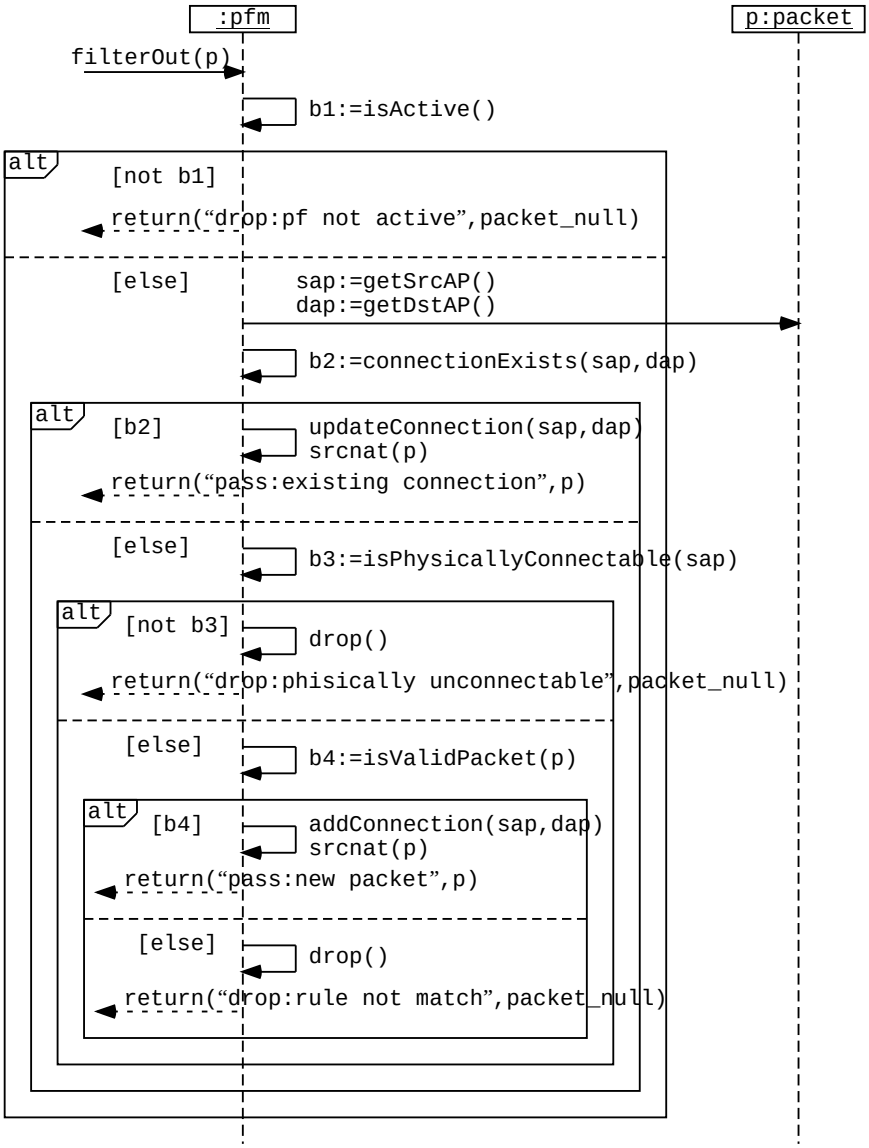

図 7.5: pfm::filterOut()

7.2.2　pfm.filterIn()

pfm クラスのメソッド filterIn() のシーケンス図を図 7.6 に示す．このメソッドは入力パケット p に対し，インバウンドフィルタを適用する．

まず，isActive() により，パケットフィルタが作動中であるかをチェックする．作動中であれば，パケット p のメソッド getDstAP() により，宛先アドレス dap を取得する．次に，dap に対応する宛先アドレス変換規則が存在するかどうかをメソッド dstnatRuleExists() によりチェックする．変換規則が存在しなければ，新しい接続を張ろうとするパケットとみなし，通過を拒否する．存在すれば，メソッド dstnat() により，宛先アドレス変換を行う．次に，パケット p のメソッド getSrcAP(), getDstAP() により，送信元アドレス sap，宛先アドレス dap をそれぞれ取得する．ここでの dap の値は宛先アドレス変換後の値であり，内部アドレスとなっている．この 2 つのアドレス間に接続が存在するかどうかをメソッド connectionExists() によりチェックする．接続が存在すれば，アウトバウンドパケットに対するリプライのパケットであるとみなし，通過を許可する．このとき，メソッド updateConnection() により接続の更新を行い，パケット p を出力する．存在しなければ，新しい接続を張ろうとするパケットであるとみなし，拒否する．

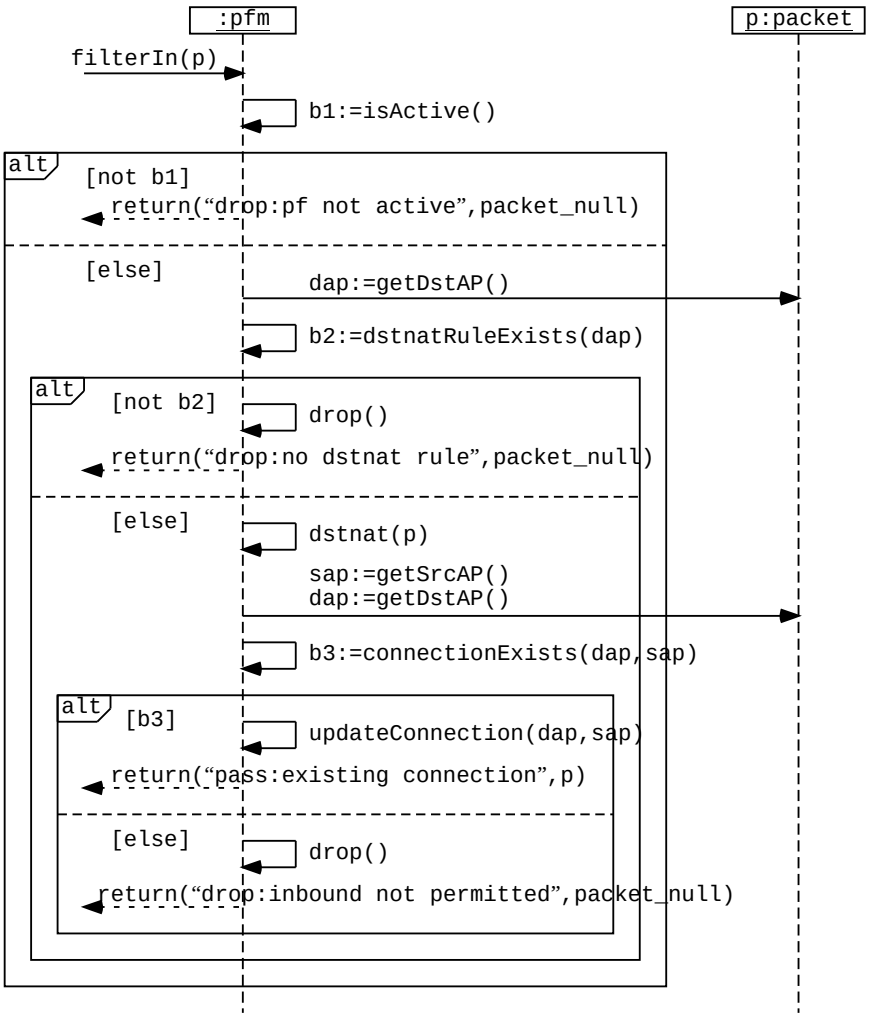

図 7.6: pfm::filterIn()

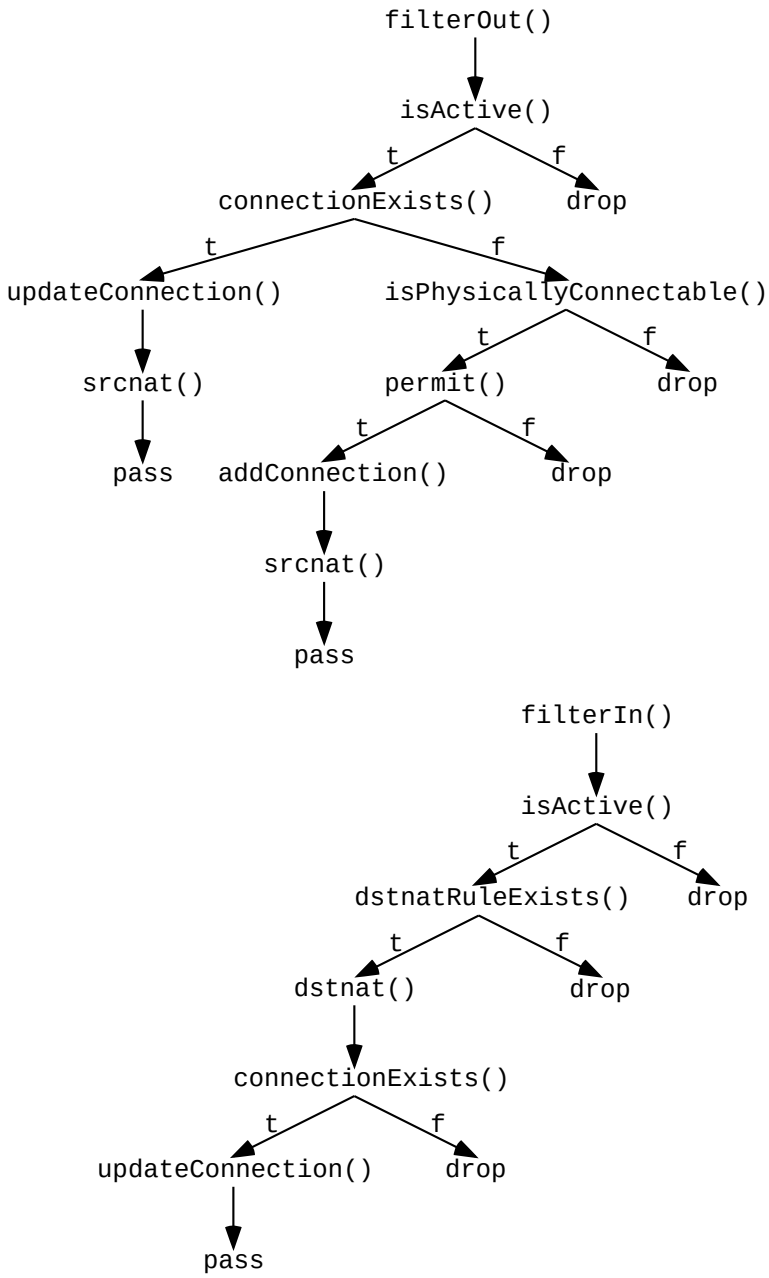

図 7.7: filterOut(), filterIn() の制御フロー

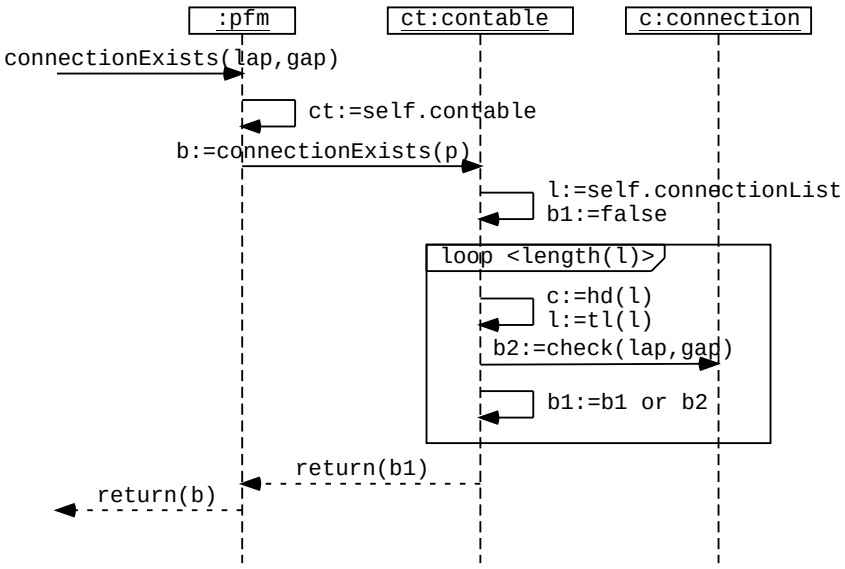

図 7.8: pfm::connectionExists()

7.2.3 pfm.connectionExists()

pfm クラスのメソッド connectionExists() のシーケンス図を図 7.8 に示す．このメソッドは，入力で与えられる内部アドレス lap と外部アドレス gap に対応する接続が存在するかどうかをチェックするメソッドである．

実際の処理は，リンクする contable オブジェクトのメソッド connectionExists() に委譲する．このメソッドでは，属性 connectionList に格納される個々の接続について，その内部アドレス，外部アドレスをチェックするために，ループ構造を用いている．ループに入る前に，接続リストを l とし，b1 を false とする．ループは l の長さだけ繰り返す．ループ内では，まず，l の先頭要素 hd(l) を c とし，先頭要素を除いた残りのリスト tl(l) を l とする．次に，メソッド check() により，c の内部アドレスと外部アドレスがそれぞれ lap, gap に一致するかどうかの真偽値 b2 を取得する．最後に b1 と b2 の論理和を b1 とし，ループの先頭に戻る．ループ終了後，b1 の値は，l の中に 1 つでも lap と gap に対応する接続が存在すれば true となっている．最後にこの値を出力し，終了する．

7.2.4 pfm.isPhysicallyConnectable()

pfm クラスのメソッド isPhysicallyConnectable() のシーケンス図を図 7.9 に示す．このメソッドは，入力される送信元アドレス sap に対して新しい接続を張るための物理容量があるかどうかをチェックする．ここでチェックすることは，接続表がいっぱいでないかどうかと，sap に対する送信元アドレス変換が存在するか，または，ポートに空きがあるかである．それぞれ，メソッド connectionIsFull(), srcnatRuleExists(), portIsFull()

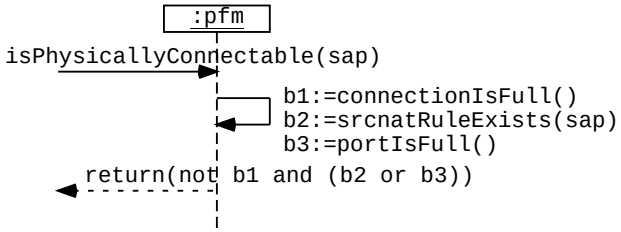

図 7.9: pfm::isPhysicallyConnectable()

によってチェックする．

7.2.5 contable.getConnection()

contable クラスのメソッド getAdmin() のシーケンス図を図 7.10 に示す．このメソッドは，入力で与えられる内部アドレス lap，外部アドレス gap を持つ接続オブジェクトを取得するメソッドである．存在しなければ connection_null を返す．

まず，属性として保持する接続リスト connectionList を l1 とし，l2 を [] とする．次にループを l1 の長さだけ繰り返す．ループ内では，まず，l1 の先頭要素 hd(l1) を c とし，先頭を除いた残りのリスト tl(l1) を l1 とする．c について，メソッド check() により，内部アドレスと外部アドレスがそれぞれ lap と gap に一致するかどうかをチェックする．一致すれば，c を l2 に格納し，ループの先頭に戻る．ループ終了後，l2 には lap, gap に一致する接続が格納されている（実際にはたかだか 1 つだけ格納される）．l2 に要素が格納されているならば，その先頭要素を出力し，空であれば，connection_null を出力する．

7.2.6 pfm.srcnat()

pfm クラスのメソッド srcnat() のシーケンス図を図 7.11 に示す．このメソッドは入力パケット p に対し，送信元アドレス変換を行うメソッドである．

まず，パケット p のメソッド getSrcAP() により，送信元アドレス sap1 を取得する．次に，nattable オブジェクト nt のメソッド srcnat() により，内部アドレス sap1 に対する変換先の公開アドレス sap2 を取得する（7.2.7 参照）．最後に，パケット p のメソッド setSrcAP() により，sap2 を送信元アドレスに書き込んで終了する．

7.2.7 nattable.srcnat()

nattable クラスのメソッド srcnat() のシーケンス図を図 7.12 に示す．このメソッドは入力内部アドレス sap に対し，変換先の公開アドレスを返すメソッドである．

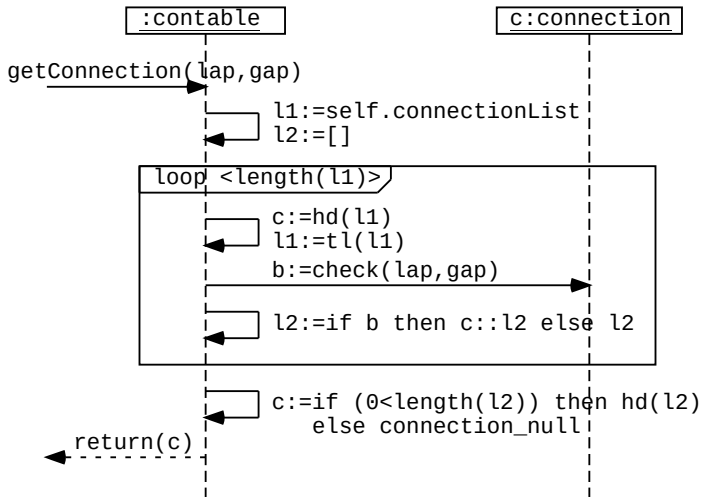

図 7.10: contable::getConnection()

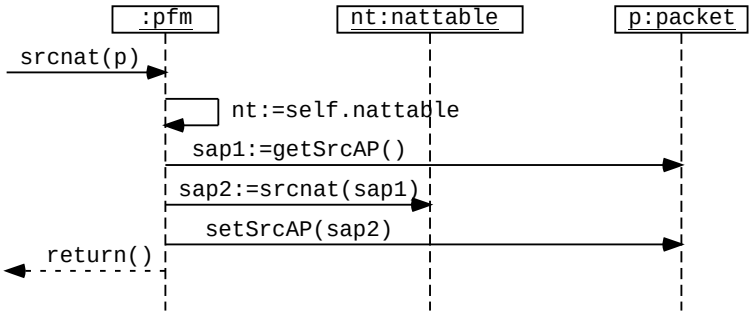

図 7.11: pfm::srcnat()

まず，メソッド srcnatRuleExists() により，アドレス sap に対応する送信元アドレス変換規則が存在するかどうかをチェックする．存在すれば，メソッド srcnat1() により sap に対応する変換先アドレス sap1 を取得し，出力する．存在しなければ，変換規則を生成を行う．変換規則の生成にはポートを 1 つ消費するため，まず，メソッド isFull() により，未使用ポートが存在するかどうかをチェックする．存在すれば，メソッド addNatrule() により，sap に対応する変換規則を生成し（7.2.8 参照），srcnat1() により変換を行い，変換後のアドレス sap1 を出力する．存在しなければ，変換規則を生成することが物理的に不可能であるため，変換を行わず，sap をそのまま出力する．

メソッド srcnat1() は，送信元アドレス変換の具体的な処理を実装している．まず，属性として保持している natrule オブジェクトのリストを l1 とし，l2 を [] とする．次に，ループを l1 の長さだけ繰り返す．ループ内では，まず，l1 の先頭要素 hd(l1) を nr とし，先頭を除いた残りのリスト tl(l1) を l1 とする．nr について，メソッド checkLocalAP()

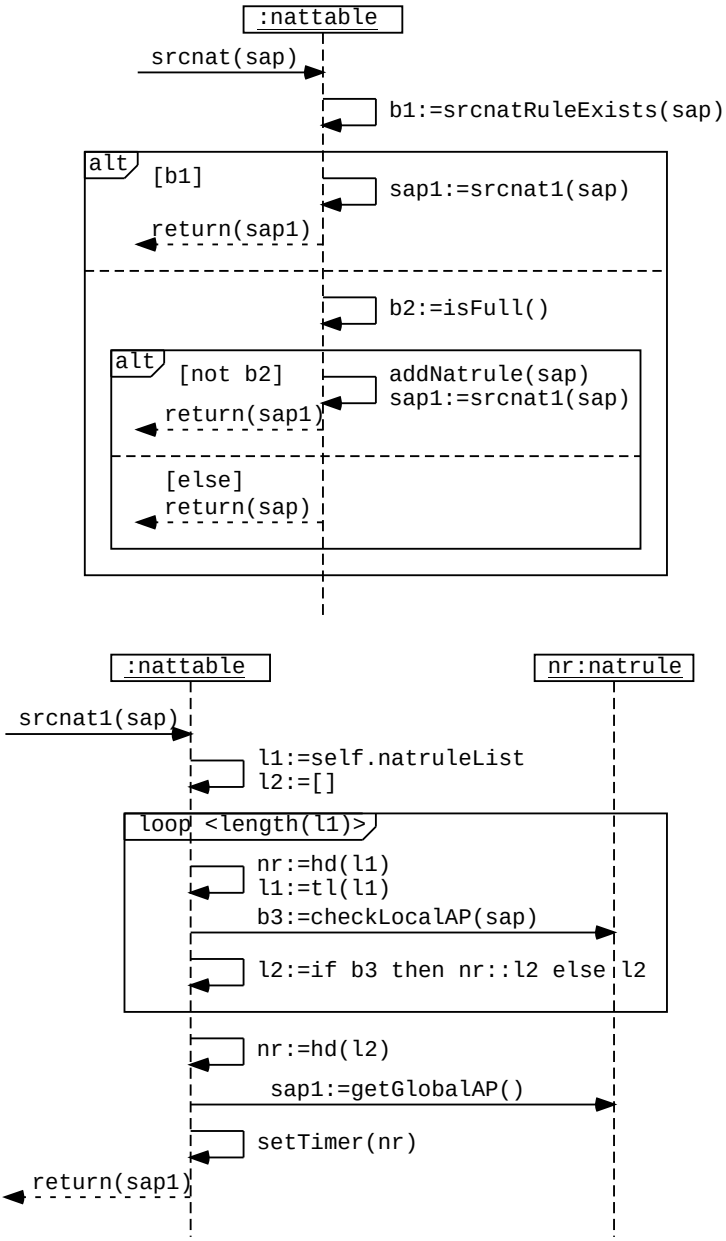

図 7.12: nattable::srcnat()

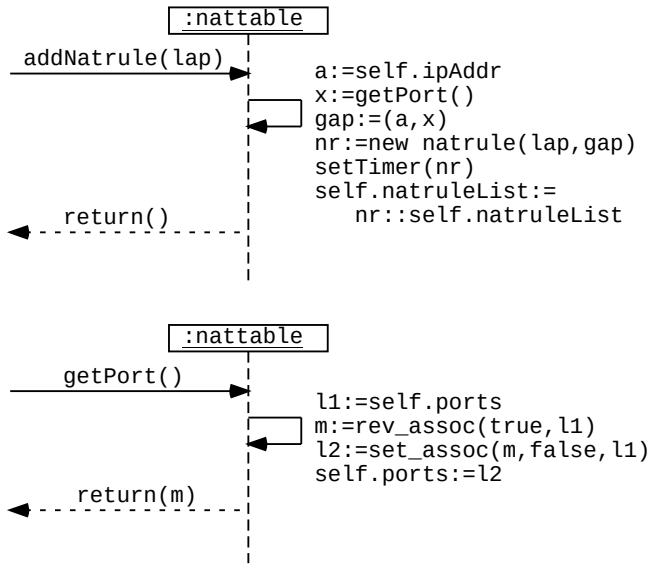

図 7.13: nattable::addNatrule()

により，nr の内部アドレス（変換元アドレス）が sap と一致するかどうかをチェックする．一致すれば，nr を l2 に格納し，ループの先頭に戻る．ループ終了後，l2 には sap に対応する変換規則が格納されている（必ず 1 つだけ格納されている）．このリストの先頭要素を nr とし，メソッド getGlobalAP() により，公開アドレス（変換先アドレス）を取得し，sap1 とする．最後に，メソッド setTimer() により，変換規則のタイムアウト時間を初期化し，sap1 を出力する．

7.2.8 nattable.addNatrule()

nattable クラスのメソッド addNatrule() のシーケンス図を図 7.13 に示す．このメソッドは，入力内部アドレス lap に対する新しい変換規則を生成するメソッドである．

まず，属性として保持している公開 IP アドレスを a とする．次に，メソッド getPort() により，未使用ポート x を取得する．これらのペア (a,x) を gap とし，入力内部アドレス lap に対応する公開アドレスとする．次に lap, gap をコンストラクタの引数として，変換規則である natrule オブジェクト nr を生成する．次に，メソッド setTimer() により，変換規則 nr のタイムアウト時間を初期化する．最後に，生成されたオブジェクト nr を属性 natruleList に追加して終了する．

メソッド getPort() は，未使用ポート番号を取得すると同時に，取得したポートの状態を使用中にするメソッドである．まず，属性 ports を l1 とする．l1 はポート番号とその使用状態を表す真偽値のペアのリストである．次に，未使用ポート番号をリスト関数 rev_assoc(true,l1) により取得する．この関数により得られる値 m は，l1 の要素を先

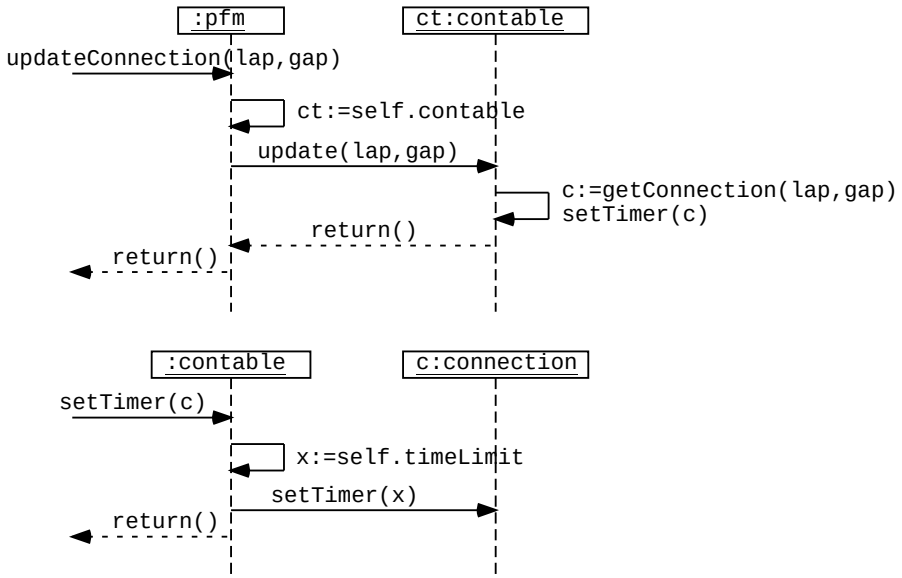

図 7.14: pfm::updateConnection()

頭から見ていったときに最初に使用状態が true，つまり未使用となっているペアのポート番号である．次に，ポート m の使用状態をリスト関数 set_assoc(m,false,l1) により使用中に変更する．この関数により得られるリスト l2 は，l1 の要素の中で，ポート番号が m となっているペアの使用状態を false にした値である．最後に，l2 を属性 ports に格納し，ポート番号 m を出力する．

7.2.9 pfm.updateConnection()

pfm クラスのメソッド updateConnection() のシーケンス図を図 7.14 に示す．このメソッドは，入力で与えられる内部アドレス lap と外部アドレス gap に対応する接続のタイムアウト時間を初期化するメソッドである．

実際の処理はリンクする contable オブジェクトのメソッド update() に委譲する．update() 内部では，まず，メソッド getConnection() により，アドレス lap と gap に対応する接続オブジェクト c を取得する (7.10 参照)．次に，メソッド setTimer() により，c のタイムアウト時間を初期化する．これは，図の下段のように，接続タイムアウト時間を表す属性 timeLimit を c のメソッド setTimer() に渡すことにより行う．

7.2.10 pfm.addConnection()

pfm クラスのメソッド addConnection() のシーケンス図を図 7.15 に示す．このメソッドは，入力内部アドレス lap と外部アドレス gap の間に接続を生成するメソッドである．

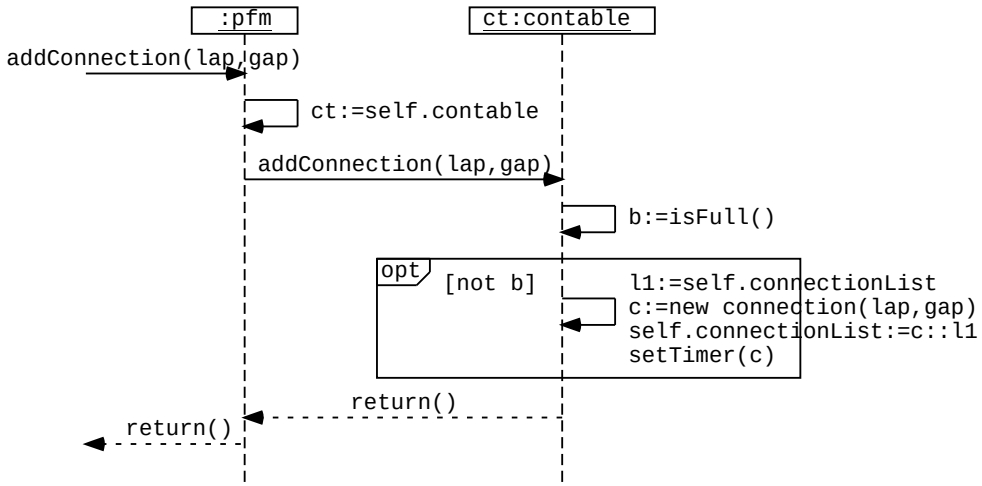

図 7.15: pfm::addConnection()

実際の処理は，リンクする contable オブジェクトを ct のメソッド addConnection() に委譲する．このメソッドでは，メソッド isFull() により接続がいっぱいでないかどうかをチェックし，空きがあるときに限り，connection オブジェクトを．lap と gap を引数として生成する．その後，生成されたオブジェクト c を属性 connectionList に追加し，メソッド setTimer() により，接続 c のタイムアウト時間を初期化して終了する．

7.2.11 pfm.isValidPacket()

pfm クラスのメソッド isValidPacket() のシーケンス図を図 7.16 に示す．このメソッドは入力パケット p をフィルタルールに照合し，通過許可判定を行うメソッドである．

まず，メソッド isOutbound() により，パケット p がアウトバウンドパケットであるかどうかをチェックする．このメソッドの内部では，図の中段のように，パケット p の送信元 IP アドレス sa，宛先アドレス da がそれぞれ，プライベートアドレス，公開アドレスであり，かつ，da がメソッド getIpAddr() により取得されるファイアウォールのアドレス x でないかをチェックする．関数 isPrivate は入力がこのモデルにおけるプライベートアドレスの予約範囲である 0..99 の値であれば真となる．

次に，リンクする frule オブジェクトのメソッド check() によりパケット p の 5 つのヘッダ情報がフィルタルールに合致するかどうかをチェックする．このメソッドの内部では，パケット p の送信元 IP アドレス sa，宛先 IP アドレス da，送信元ポート番号 sp，宛先ポート番号 dp，プロトコル pr が許可される値であるかどうかを，それぞれ，メソッド checkSrcAddr(), checkDstAddr(), checkSrcPort(), checkDstPort(), checkProtocol() によりチェックする．メソッド checkSrcAddr() の内部では，図の下段に示すように，入力アドレス x が，許可される送信元 IP アドレスのリスト srcAddrTable の要素であるかをリスト関数 mem

を用いてチェックする．その他の 4 つのメソッドについても同様の実装となっている．

最後に，両方のチェック結果の論理積を出力する．

7.2.12　pfm.incSec()

pfm クラスのメソッド incSec() のシーケンス図を図 7.17 に示す．このメソッドは時間を入力値 n 秒だけ進めるメソッドである．内部的には，時間を 1 秒進ませるメソッド incSec1() を n 回繰り返すことにより実現している．

incSec1() は，まず，contable オブジェクト ct に対し，メソッド decTimer() により，すべての接続のタイムアウト残り時間を 1 減算する（7.2.13 参照）．次に，nattable オブジェクト nt に対し，メソッド decTimer() により，すべてのアドレス変換規則のタイムアウト残り時間を 1 減算する．最後に，doscounter オブジェクト dc に対し，メソッド reset() により，カウンタをリセットする．

7.2.13　contable.decTimer()

contable クラスのメソッド decTimer() のシーケンス図を図 7.18 に示す．このメソッドは，すべての接続のタイムアウト残り時間を 1 減算し，その結果 0 となった接続を削除するメソッドである．

まず，属性として保持する接続リスト connectionList を l1 とし，l2 を [] とする．次にループを l1 の長さだけ繰り返す．ループ内では，まず，l1 の先頭要素 hd(l1) を c とし，先頭を除いた残りのリスト tl(l1) を l1 とする．c について，メソッド decTimer() により，タイムアウト残り時間を 1 減算する．このメソッドの出力が false ならば，つまり，減算後，残り時間が 0 でないならば，l2 に格納し，ループの先頭に戻る．ループ終了後，l2 には，まだタイムアウトしていない接続が格納されている．最後に，l2 を属性 connectionList に格納し，終了する．

nattable クラスのメソッド decTimer() も同様な方法で定義される．

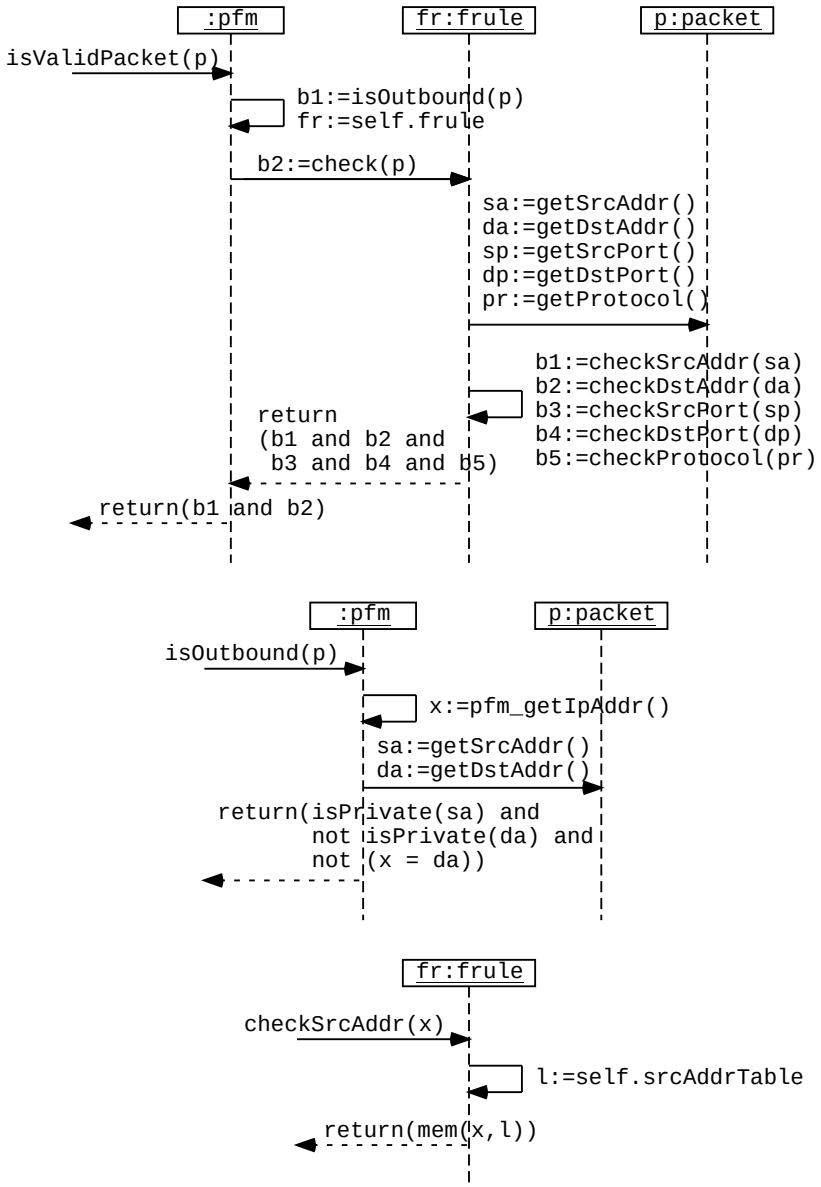

図 7.16: pfm::isValidPacket()

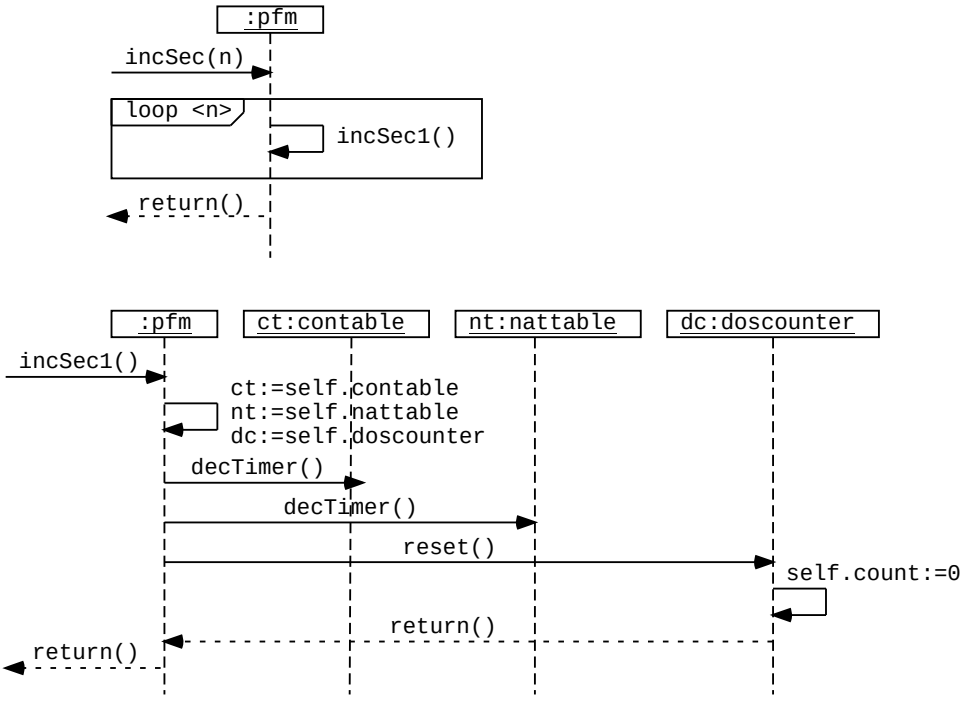

図 7.17: pfm::incSec()

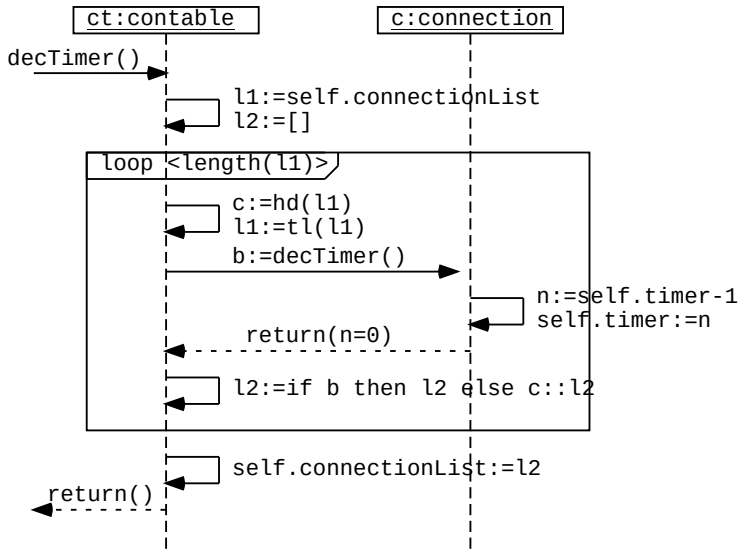

図 7.18: contable::decTimer()

第8章　MLにおけるシミュレーション実行

MLにおけるシミュレーション実行の様子を以下に示す．

まず，パケットフィルタオブジェクトを生成する．

```
- val (pfm,s) = new_pfm store_emp
> val pfm = <pfm> : pfm
  val s = <store> : store
```

次に，各種コンフィギュレーションの設定を行う．

```
- val (_,s) = pfm_setIpAddr pfm 200 s;  (1)
- val (_,s) = pfm_setPorts pfm [1200,1210,1220] s;  (2)
- val (_,s) = pfm_setMaxConnectionSize pfm 5 s;  (3)
- val (_,s) = pfm_setConnectionTime pfm 300 s;  (4)
- val (_,s) = pfm_setNatruleTime pfm 500 s;  (5)
- val (_,s) = pfm_setDosThreshould pfm 5 s;  (6)
```

(1) ファイアウォールの公開アドレスを 200 とする．(2) 公開ポートを 1200, 1210, 1220 の 3 つとする．(3) 最大接続数を 5 とする．(4) 接続タイムアウト時間を 300 とする．(5) アドレス変換規則のタイムアウト時間を 500 とする．(6) DoS カウンタの閾値を 5 とする．

次に，フィルタルールの設定を行う．

```
- val (_,s) = pfm_setFilterRules pfm SRCADDR [10,20,30] s;  (1)
- val (_,s) = pfm_setFilterRules pfm SRCPORT [1050,1070,1100] s;  (2)
- val (_,s) = pfm_setFilterRules pfm DSTADDR [250] s;  (3)
- val (_,s) = pfm_setFilterRules pfm DSTPORT [80] s;  (4)
- val (_,s) = pfm_setFilterRules pfm PROTOCOL [TCP] s;  (5)
```

(1) 許容送信元 IP アドレスを 10, 20, 30 とする．(2) 許容送信元ポートを 1050, 1070, 1100 とする．(3) 許容宛先 IP アドレスを 250 とする．(4) 許容宛先ポートを 80 とする．(5) 許容プロトコルを TCP とする．

ここで，パケットフィルタを起動し，パケット送受信を開始する．

```
- val (b,s) = pfm_start pfm s;
> val b = true : bool
  val s = <store> : store
```

まず，アウトバウンドフィルタの動作を確認するために，内部ホストから外部ホストへパケットを送信してみる．パケットの生成は次のように行う．

```
- val (p,s) = new_packet 20 1070 250 80 TCP s;
> val p = <packet> : packet
  val s = <store> : store
```

これは送信元アドレスが (20,1070)，宛先アドレスが (250,80)，プロトコルが TCP のパケットである．

このパケットに対しアウトバウンドフィルタ pfm_filterOut を適用する．

```
- val (msg,p,s) = pfm_filterOut pfm p s;
> val msg = "pass: new connection" : string
  val p = <packet> : packet
  val s = <store> : store
```

入力パケットはフィルタルールを満たすので，通過が許可される．出力メッセージ msg は，入力パケットを新しい接続のパケットとして通過許可したことを表している．実際に次のように接続表を表示することにより，新しい接続が生成されていることがわかる．

```
- pfm_getConnections pfm s;
> val it = [((20, 1070), (250, 80), 300)] :
    ((int * int) * (int * int) * int) list
  val s = <store> : store
```

300 という値は，この接続のタイムアウト残り時間である．

また，アドレス変換表は次のようになっている．

```
- pfm_getNatrules pfm s;
> val it = [((20, 1070), (200, 1200), 500)]
    : (((int * int) * (int * int)) * int) list
```

これは，接続が生成された際に動的に生成された変換規則であり，内部アドレス (20, 1070) をファイアウォールの公開アドレス (200,1200) に変換するという規則である．この規則のタイムアウト残り時間は 500 である．

送出されたパケットの送信元アドレスが確かに変換されていることは次のように確認することができる．

```
- packet_getInfo p s;
> val it = ((200, 1200), (250, 80), 0)
    : (int * int) * (int * int) * int
```

これは，パケット p の送信元アドレスが (200,1200)，宛先アドレスが (250,80)，プロトコルが 0（変数 TCP の値）であることを意味する．したがって，確かに送信元アドレスが最初の (20,1070) から公開アドレス (200,1200) に変換されている．

一方，次のアウトバウンドパケットは通過拒否される．

```
- val (p,s) = new_packet 20 1070 250 53 TCP s;
- val (msg,p,s) = pfm_filterOut pfm p s;
> val msg = "drop: rule not match" : string
  val p = <packet> : packet
  val s = <store> : store
```

このパケットは宛先ポートが 53 となっており，フィルタルールを満たさないために拒否される．

次に，インバウンドフィルタの動作を確認するために，先程通過したパケットに対するリプライのパケットを生成し，インバウンドフィルタ pfm_filterIn を適用する．

```
- val (p,s) = new_packet 250 80 200 1200 TCP s;
- val (msg,p,s) = pfm_filterIn pfm p s;
> val msg = "pass: existing connection" : string
  val p = <packet> : packet
  val s = <store> : store
```

このパケットは既存の接続に属すものであるから，無条件に通過が許可される．この出力パケットの内容は次のようになっている．

```
- packet_getInfo p s;
> val it = ((250, 80), (20, 1070), 0)
    : (int * int) * (int * int) * int
```

インバウンド送信ではアウトバウンド送信の際に生成された変換規則を用いて，宛先アドレスを (200,1200) から (20,1070) に変換する．結果としてリプライがアウトバウンド送信を行った内部アドレスに届くことになる．

一方，次のインバウンドパケットは通過拒否される．

```
- val (p,s) = new_packet 250 80 200 1210 TCP s;  (3)
- val (p,msg,s) = pfm_filterIn pfm p s;
> val p = <packet> : packet
  val msg = "drop: no dstnat rule" : string
  val s = <store> : store
```

このパケットは外部アドレス (250,80) からファイアウォールの公開アドレス (200,1210) へ送信されたものである．このパケットが拒否されるのは，(200,1210) に対する宛先アドレス変換規則が定義されていないためである．仮に，これが許可されたアウトバウンドパケットのリプライのパケットであればアウトバウンド送信時に (200,1210) に対応する変換規則が定義されていたはずである．したがって，このパケットは新しいインバウンド送信とみなされ，通過拒否される．

ここで，タイムアウト時間の動作を確かめるため，時間を 100 秒経過させ，接続表，アドレス変換表を表示してみる．

```
- val s = pfm_incSec pfm 100 s;
> val s = <store> : store
- pfm_getConnections pfm s;
> val it = [((20, 1070), (250, 80), 200)]
    : ((int * int) * (int * int) * int) list
- pfm_getNatrules pfm s;
> val it = [((20, 1070), (200, 1200), 400)]
    : ((int * int) * (int * int) * int) list
```

pfm_incSec は時間を引数で与えられる秒数だけ進める関数である．それぞれのタイムアウト時間が 100 秒減少し，200, 400 となっていることがわかる．タイムアウト時間は，接続の場合，その接続に属すパケット送受信があったとき，変換規則の場合，その変換規則が使用されたとき，再設定される．そこで，リプライのアウトバウンドパケットを次のように送信する．

```
- val (p,s) = new_packet 20 1070 250 80 TCP s;
- val (p,msg,s) = pfm_filterOut pfm p s;
val p = <packet> : packet
val msg = "pass: existing connection" : string
val s = <store> : store
```

これは既存の接続に属すアウトバウンドパケットであるから，フィルタルールに無条件に通過する．

ここで，接続表，アドレス変換表を表示すると，確かにタイムアウト時間がもとの 300, 500 に再設定されていることが分かる．

```
- pfm_getConnections pfm s;
> val it = [((20, 1070), (250, 80), 300)]
    : ((int * int) * (int * int) * int) list
- pfm_getNatrules pfm s;
> val it = [((20, 1070), (200, 1200), 500)]
    : ((int * int) * (int * int) * int) list
```

次の 3 つのパケット送受信はいずれも許可される．

```
- val (p,s) = new_packet 10 1050 250 80 TCP s;  (1)
- val (p,msg,s) = pfm_filterOut pfm p s;
- val (p,s) = new_packet 30 1100 250 80 TCP s;  (2)
- val (p,msg,s) = pfm_filterOut pfm p s;
- val s = pfm_incSec pfm 100 s;
- val (p,s) = new_packet 250 80 200 1220 TCP s;  (3)
- val (p,msg,s) = pfm_filterIn pfm p s;
```

(1) は新しい接続のパケットであり，フィルタルールを満たすので通過する（この時点で，図 6.1 の状態となる）．(2) も同様に新しい接続のパケットであり，フィルタルールを満たすので通過する（図 6.2）．(3) は (2) に対するリプライであり，直前に 100 秒経過しているが，(2) の接続はまだタイムアウトしていない（残り時間 200）ため，通過する（図 6.3）．

この時点で接続表，アドレス変換表は次のようになっている．

```
- pfm_getConnections pfm s;
> val it =
    [((30, 1100), (250, 80), 300), ((10, 1050), (250, 80), 200),
     ((20, 1070), (250, 80), 200)]
    : ((int * int) * (int * int) * int) list
- pfm_getNatrules pfm s;
> val it =
    [((30, 1100), (200, 1220), 500), ((10, 1050), (200, 1210, 400)),
     ((20, 1070), (200, 1200), 400)]
    : ((int * int) * (int * int) * int) list
```

ここで，時間を 200 秒経過させ，残り時間が 200 となっている 2 つの接続を切断する．

```
- val s = pfm_incSec pfm 200 s;
- pfm_getConnections pfm s;
> val it = [((30, 1100), (250, 80), 100)] :
    ((int * int) * (int * int) * int) list
- pfm_getNatrules pfm s;
> val it =
    [((30, 1100), (200, 1220), 300), ((10, 1050), (200, 1210), 200),
     ((20, 1070), (200, 1200), 200)]
    : ((int * int) * (int * int) * int) list
```

この状態から次のように様々な場合のパケット送受信を行う．

```
- val (p,s) = new_packet 10 1100 250 80 TCP s;  (1)
- val (p,msg,s) = pfm_filterOut pfm p s;
val p = <packet> : packet
val msg = "drop: physically unconnectable" : string
val s = <store> : store
- val (p,s) = new_packet 20 1070 250 80 TCP s;  (2)
- val (p,msg,s) = pfm_filterOut pfm p s;
val p = <packet> : packet
val msg = "pass: new connection" : string
val s = <store> : store
- val (p,s) = new_packet 250 80 200 1210 TCP s;  (3)
- val (p,msg,s) = pfm_filterIn pfm p s;
val p = <packet> : packet
val msg = "drop: inbound not permit" : string
val s = <store> : store
```

(1) は内部アドレス (10,1100) からの新しい接続のパケットである．このアドレスに対する変換規則は定義されていないので新しく生成されなければならないが，現時点で 3 つのポート 1050, 1070, 1100 すべてが占有されているため，変換規則が生成できない状態である．したがって，物理的に接続不可能というメッセージとともに通過拒否される．(2) も同様に内部アドレス (20,1070) からの新しい接続のパケットであるが，この場合，変換規則がすでに定義されている（変換表の最後の要素）ため，新たに規則を生成する必要がない．したがって，物理的容量の問題はクリアし，接続可能となる．(3) は，外部アドレス (250,80) からファイアウォールの公開アドレス (200,1210) へのインバウンド送信である．この場合，(200, 1210) に対する宛先アドレス変換規則は定義されているが，変換先の内部アドレス (10,1050) との間に接続が存在しないということで新しいインバウンド送信とみなし，通過拒否される．

ここで，DoS カウンタの動作を確認するために，次の 3 つの不正インバウンドパケットを受信する．

```
- val (p,s) = new_packet 250 80 200 1210 TCP s;  (4)
- val (p,msg,s) = pfm_filterIn pfm p s;
val p = <packet> : packet
val msg = "drop: inbound not permit" : string
val s = <store> : store
- val (p,s) = new_packet 250 80 200 1210 TCP s;  (5)
- val (p,msg,s) = pfm_filterIn pfm p s;
val p = <packet> : packet
val msg = "drop: inbound not permit" : string
val s = <store> : store
- val (p,s) = new_packet 250 80 200 1210 TCP s;  (6)
- val (p,msg,s) = pfm_filterIn pfm p s;
val p = <packet> : packet
val msg = "drop: inbound not permit" : string
val s = <store> : store
```

この時点で，この秒間における拒否されたパケットの数が，(4)(5)(6) と，前の (1)(3) を合わせて DoS カウンタ閾値である 5 に達したため，次のように，警告フラグが立っているのが確認できる．

```
- pfm_getDosAlert pfm s;
> val it = true
```

最後に，パケットフィルタを停止する．停止時には接続表，変換表がリセットされる．

```
- val (b,s) = pfm_stop pfm s;
> val b = true : string
  val s = <store> : store
- pfm_getConnections pfm s;
> val it = [] : ((int * int) * (int * int) * int) list
- pfm_getNatrules pfm s;
> val it = [] : ((int * int) * (int * int) * int) list
```

第9章　HOLにおける定理証明

本章では，pfm クラスのメソッド filterOut() に関する事前事後条件の証明と，pfm クラスに関する不変表明の証明を行う．

メソッド事前事後条件の証明では，パケットが通過許可される必要十分条件（図 9.1A）の証明と，送信元アドレス変換の正当性，つまり，パケットの送信元アドレスが必ずファイアウォールの公開アドレスに変換されること（図 9.1B）の証明を行う[1]．

不変表明の証明では，接続表とアドレス変換表の間の一貫性を証明する．この不変表明は，接続が存在するならば必ずその内部アドレスに対応する変換規則が存在するというものである（図 9.1C）．この不変表明がメソッド filterOut() と incSec() のそれぞれの適用前後で成立することを証明する．

9.1　パケット通過許可条件の証明

アウトバウンドフィルタ filterOut() によってパケットが通過許可されるための必要十分条件の証明を行う．

命題の設定

証明対象の命題は次のようになる．

```
!pfm p s.
  let (msg,p1,s1) = pfm_filterOut pfm p s in
  let sap = packet_getSrcAP p s in
  let dap = packet_getDstAP p s in
    packet_ex p s ==>  (1)
      (~(p1 = packet_null) =  (2)
      pfm_isActive pfm s /\  (3)
      (pfm_connectionExists pfm sap dap s \/  (4)
       pfm_isPhysicallyConnectable pfm sap s /\  (5)
       pfm_isValidPacket pfm p s)  (6)
```

[1] A の右辺の permit() は制約を分けて記述するために導入した架空のメソッドである．

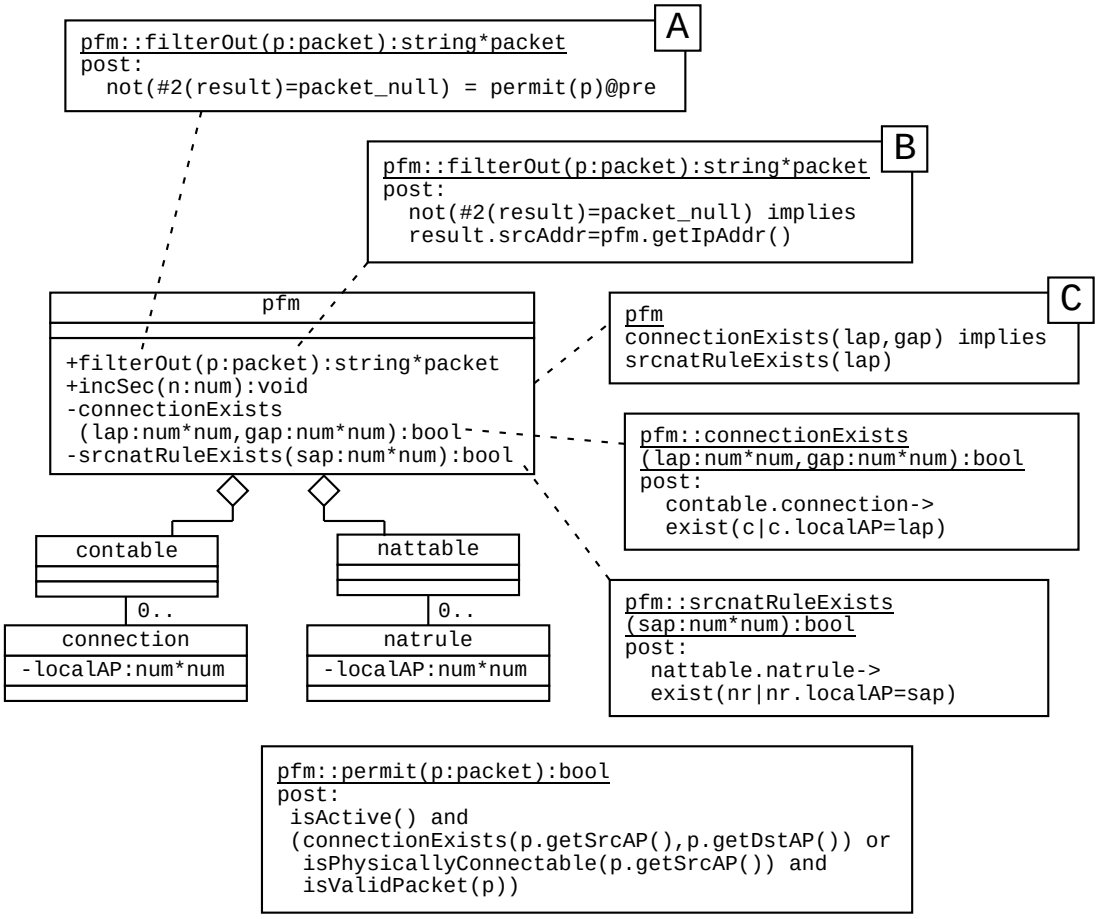

図 9.1: 検証対象の OCL による記述

メソッド pfm_filterOut は，入力パケット p を処理し，パケット p1 として出力する．パケットが拒否されるとき，p1 は packet_null となるため，結論部の左辺 (2) は，パケットが通過許可されたことを意味する．結論部の右辺は，そのための必要十分条件である．つまり，p が通過許可されることは，(3) pfm がアクティブ，かつ，(4) p の送信元アドレス sap，宛先アドレス dap に対応する接続が存在する，または，(5) 新しい sap からの接続を物理的に生成可能であり，(6) p がフィルタルールを満たすパケットである，と同値である．前提条件 (1) は，p が pfm_filterOut 適用前のストアに存在することを意味する．つまり，pfm_filterOut には，必ず生成されたパケットが入力されると仮定する．

証明

必要十分条件を ⇒) と ⇐) の 2 つに分けて証明する．

⇒) の証明

証明対象のゴールは次のようになる．

```
!pfm p s.
  let (msg,p1,s1) = pfm_filterOut pfm p s in
  let sap = packet_getSrcAP p s in
  let dap = packet_getDstAP p s in
    packet_ex p s ==>
      (~(p1 = packet_null) ==>
      pfm_isActive pfm s /\
      (pfm_connectionExists pfm sap dap s \/
       pfm_isPhysicallyConnectable pfm sap s /\ pfm_isValidPacket pfm p s)
```

まず，結論部の対偶をとる．

```
!pfm p s.
  let (msg,p1,s1) = pfm_filterOut pfm p s in
  let sap = packet_getSrcAP p s in
  let dap = packet_getDstAP p s in
    packet_ex p s ==>
      (~pfm_isActive pfm s \/
       ~pfm_connectionExists pfm sap dap s /\
       (~pfm_isPhysicallyConnectable pfm sap s \/
        ~pfm_isValidPacket pfm p s)
    ==> (p1 = packet_null))
```

ここで，トートロジー

```
|- !P Q R S. P \/ Q /\ (R \/ S) = P \/ ~P /\ Q /\ R \/ ~P /\ Q /\ ~R /\ S
```

により書き換えを行うと次のようになる．

```
!pfm p s.
  let (msg,p1,s1) = pfm_filterOut pfm p s in
  let sap = packet_getSrcAP p s in
  let dap = packet_getDstAP p s in
    packet_ex p s ==>
      (~pfm_isActive pfm s \/  (1)
       pfm_isActive pfm s /\  (2)
       ~pfm_connectionExists pfm sap dap s /\  (3)
       ~pfm_isPhysicallyConnectable pfm sap s \/  (4)
       pfm_isActive pfm s /\  (5)
       ~pfm_connectionExists pfm sap dap s /\  (6)
       pfm_isPhysicallyConnectable pfm sap s /\  (7)
       ~pfm_isValidPacket pfm p s)  (8)
       ==> (p1 = packet_null))
```

ここで，論理和により結合される (1) と (2)〜(4) と (5)〜(8) の 3 つの場合はいずれもパケットを拒否する条件，つまり，p1 を packet_null にする条件である．これは次の pfm_filterOut の定義と照らし合わせることにより確かめることができる．

```
pfm_filterOut pfm p s =
  if ~pfm_isActive pfm s then  (A)
    ("drop: pf not active", packet_null, s)  (B)
  else
    let sap = packet_getSrcAP p s in
    let dap = packet_getDstAP p s in
      if pfm_connectionExists pfm sap dap s then  (C)
        let s = pfm_srcnat pfm p s in
        let s = pfm_updateConnection pfm sap dap s in
          ("pass: existing connection", p, s)  (D)
      else if pfm_isPhysicallyConnectable pfm sap s then  (E)
        if pfm_isValidPacket pfm p s then  (F)
          let s = pfm_addConnection pfm sap dap s in
          let s = pfm_srcnat pfm p s in
            ("pass: new connection", p, s)  (G)
        else
          let s = pfm_drop pfm s in
            ("drop: rule not match", packet_null, s)  (H)
```

```
      else
        let s = pfm_drop pfm s in
          ("drop: phisically unconnectable", packet_null, s)  (I)
```

まず，(1) の場合について，(1) は (A) を満たすため，(B) の packet_null が出力される．次に，(2)〜(4) の場合について，(2) は (A) を否定し，(3) は (C) を否定し，(4) は (E) を否定するため，(I) にたどり着き，packet_null が出力される．最後に，(5)〜(8) の場合について，(5) は (A) を否定し，(6) は (C) を否定し，(7) は (E) を満たし，(8) は (F) を否定するため，(H) にたどり着き，packet_null が出力される．

以上 3 つの場合からいずれも p1=packet_null が導出されるため，ゴールが証明される．

⇐) の証明

証明対象のゴールは次のようになる．

```
!pfm p s.
  let (msg,p1,s1) = pfm_filterOut pfm p s in
  let sap = packet_getSrcAP p s in
  let dap = packet_getDstAP p s in
    packet_ex p s ==>
      (pfm_isActive pfm s /\
       (pfm_connectionExists pfm sap dap s \/
        pfm_isPhysicallyConnectable pfm sap s /\ pfm_isValidPacket pfm p s)
       ==> ~(p1 = packet_null)
```

まず，トートロジー

```
|- !P Q R S. P /\ (Q \/ R /\ S) = P /\ Q \/ P /\ ~Q /\ R /\ S
```

により書き換えを行うと次のようになる．

```
!pfm p s.
  let (msg,p1,s1) = pfm_filterOut pfm p s in
  let sap = packet_getSrcAP p s in
  let dap = packet_getDstAP p s in
    packet_ex p s ==>
      (pfm_isActive pfm s /\  (1)
       pfm_connectionExists pfm sap dap s \/  (2)
       pfm_isActive pfm s /\  (3)
       ~pfm_connectionExists pfm sap dap s /\  (4)
       pfm_isPhysicallyConnectable pfm sap s /\  (5)
       pfm_isValidPacket pfm p s)  (6)
       ==> ~(p1 = packet_null)
```

ここで，論理和により結合される (1)(2) と (3)〜(6) の 2 つの場合はいずれもパケットを通過させる条件，つまり，p1 を p にする条件である．これは前の証明と同様，pfm_filterOut の定義と照らし合わせることにより確かめることができる．

まず，(1)(2) の場合について，(1) は (A) を満たし，(2) は (C) を満たすため，(D) にたどり着き，p が出力される．次に，(3)〜(6) の場合について，(3) は (A) を満たし，(4) は (C) を否定し，(5) は (E) を満たし，(6) は (F) を満たすため，(G) にたどり着き，p が出力される．

以上 2 つの場合からいずれも p1=p が導出されるため，ゴールは次のように簡単化される．

```
packet_ex p s ==> ~(p = packet_null)
```

オブジェクト指向理論には，「ストアに存在するオブジェクトは NULL ではない」という定理が存在し，これによりゴールが証明される．

以上，⇒) と ⇐) の 2 つの証明を合わせて目的の命題が証明される．

9.2　アドレス変換正当性の証明

アウトバウンド送信において通過許可されたパケットには送信元アドレス変換が行われる．図 9.2 に示すように，変換後，送信元 IP アドレスはファイアウォールの公開 IP アドレスとなっていなければならない．これはプライベートアドレスが外部に漏洩しないという点でも重要である．これが満たされることをメソッド fiterOut() の事後条件として証明する．

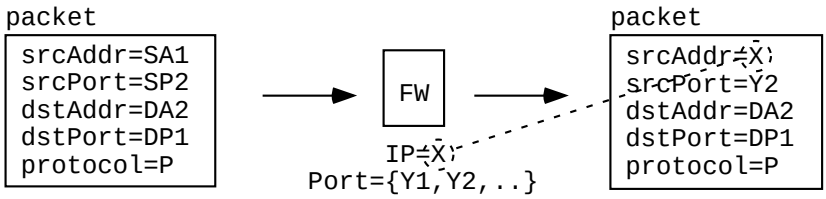

図 9.2: アドレス変換の正当性

命題の設定

証明対象の命題は次のようになる．

```
!pfm p s . let (msg,p1,s1) = pfm_filterOut pfm p s in
Pre1 pfm p s ==> ~(p1 = packet_null) ==>
   (packet_get_srcAddr p1 s1 = pfm_getIpAddr pfm s)
```

pfm_filterOut の出力について，p1 は入力パケット p のフィルタ後のパケットであり，s1 はフィルタ後のストアである．つまり命題の意味は，フィルタ後のパケットが NULL でなければ（通過許可されたパケットならば），フィルタ後のストアにおけるパケットの送信元アドレスはフィルタ前のストアにおけるパケットフィルタリングシステムの持つ公開 IP アドレスの値と等しい，となる．つまり，送信元アドレスはファイアウォールの公開アドレスに変換される．前提条件 Pre1 はこの等式が成立するための様々な条件を集めたものであり，次のように定義される．

```
|- !pfm p s. Pre1 pfm p s =
     pfm_ex pfm s /\  (1)
     packet_ex p s /\  (2)
     nattable_ex_Inv pfm s /\  (3)
     con_nat_Inv pfm s /\  (4)
     natrule_Inv pfm s  (5)
where
  |- !pfm s. nattable_ex_Inv pfm s =
       let nt = pfm_get_nattable pfm s in
         nattable_ex nt s
  |- !pfm s. con_nat_Inv pfm s =
       !lap gap. pfm_connectionExists pfm lap gap ==>
         pfm_srcnatRuleExists pfm lap s
  |- !pfm s. natrule_Inv pfm s =
       let nt = pfm_get_nattable pfm s in
       let l = nattable_get_natruleList nt s in
         !x. MEM x l ==>
           (FST (natrule_get_globalAP x s) = pfm_getIpAddr pfm s)
```

(1)(2) は，pfm オブジェクト pfm と packet オブジェクト p がフィルタ前のストア s に存在することを意味する．メソッド pfm_filterOut は必ず生成された pfm オブジェクトに対して起動され，また，packet オブジェクトも必ず生成されたものが入力されるので，これらの前提条件を導入している．

(3)(4)(5) は不変表明である．(3) は，pfm オブジェクトとリンクする nattable オブジェクトは必ずストアに存在することを意味する．ファイアウォール初期化時に pfm オブジェクトと nttable オブジェクトが生成されリンクされるが，このリンク形状はその後適用されうるいかなるメソッドによっても変化しないため，この不変表明が成立することは明らかである．(4) は，内部アドレス lap と外部アドレス gap の間に接続が存在するならば，sap に対する送信元アドレス変換が存在することを意味する．この不変表明は成立するかどうかは証明されない限り確証は得られないが，ここでは成立するものとして前提条件に含めておく．この不変表明の証明については次節で述べる．(5) は，変換表 nattable オブジェクトが保持する変換規則の要素すべてについて，その変換先 IP アドレス FST (natrule_get_globalAP x s)

がファイアウォールの公開 IP アドレス pfm_getIpAddr pfm s と等しいことを意味する．この公開 IP アドレスの値は，pfm_setIpAddr によってのみ設定され，以後に生成されるすべての変換規則の変換先 IP アドレスにはこの値が設定される．パケットフィルタが作動中はその値を変更するメソッドは存在しないため，この不変表明が成立することは明らかである．

証明

まず，前提条件~(p1 = packet_null) を 9.1 において証明した定理で書き換えることにより，パケット通過許可条件を獲得する．

```
!pfm p s.
  let (msg,p1,s1) = pfm_filterOut pfm p s in
  let sap = packet_getSrcAP p s in
  let dap = packet_getDstAP p s in
    Pre1 pfm s /\
    pfm_isActive pfm s /\
    (pfm_connectionExists pfm sap dap s \/
     pfm_isPhysicallyConnectable pfm sap s /\
     pfm_isValidPacket pfm p s) ==>
      (packet_get_srcAddr p1 s1 = pfm_getIpAddr pfm s)
```

前提条件の論理和を分解することにより，次の 2 つのサブゴールが得られる．

サブゴール (1)

```
let (msg,p1,s1) = pfm_filterOut pfm p s in
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre1 pfm s /\
  pfm_connectionExists pfm sap dap s /\ ... ==>
    (packet_get_srcAddr p1 s1 = pfm_getIpAddr pfm s)
```

サブゴール (2)

```
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre1 pfm s /\
  ~(pfm_connetionExists pfm sap dap s) /\
  pfm_isPhysicallyConnectable pfm sap s /\ ... ==>
    (packet_get_srcAddr p1 s1 = pfm_getIpAddr pfm s)
```

サブゴール (1) の証明

このサブゴールは，図 9.3 上段のような，接続がすでに存在する場合である．pfm_filterOut の定義を展開し，簡単化すると，次のようになる．

```
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre1 pfm s /\
  pfm_connectionExists pfm sap dap s /\ ... ==>
    (packet_get_srcAddr p (pfm_srcnat pfm p s) = pfm_getIpAddr pfm s)
```

「sap, dap に対応する接続が存在する」という前提条件

```
pfm_connectionExists pfm sap dap s
```

と，Pre1 に含まれる,「接続が存在するならばそれに対応する送信元アドレス変換規則が存在する」という不変表明 con_nat_Inv p s を用いて,「sap に対応する変換規則が存在する」という事実

```
pfm_srcnatRuleExists pfm sap s
```

つまり，

```
nattable_srcnatRuleExists nt sap s
```

が導出可能である．

この事実を前提条件に加え，ゴールを分解していくと次の命題が得られる．

```
let sap = packet_getSrcAP p s in
let nt = pfm_get_nattable pfm s in
let l = nattable_get_natruleList nt s in
let nr = FIRST (\x. natrule_checkLocalAP x sap s) l in
  nattable_srcnatRuleExists nt sap s /\ ... ==>
    (packet_get_srcAddr p
      (packet_set_srcAddr p
        (FST (natrule_get_globalAP nr s)) s) = pfm_getIpAddr pfm s)
```

ここで，packet_set_srcAddr は，pfm_srcnat の内部で呼び出されていた項である．つまり，送信元アドレス変換により

```
FST (natrule_get_globalAP nr s)
```

という値が，パケットの送信元アドレスに設定されていることが分かる．したがって，packet_get_srcAddr により取得される値は当然，その設定された値であるから，ゴールは次のように簡単化される．

```
let sap = packet_getSrcAP p s in
let nt = pfm_get_nattable pfm s in
let l = nattable_get_natruleList nt s in
let nr = FIRST (\x. natrule_checkLocalAP x sap s) l in
  nattable_srcnatRuleExists nt sap s /\ ... ==>
    (FST (natrule_get_globalAP nr s) = pfm_getIpAddr pfm s)
```

この命題は,「sap に対応する送信元アドレス変換規則 nr について，その変換先 IP アドレスの値は，ファイアウォールの公開 IP アドレスに等しい」を意味する．これを証明するには，Pre1 に含まれる,「変換表に含まれる変換規則すべてについて，その変換先 IP アドレスがファイアウォールの公開 IP アドレスと等しい」を意味する不変表明 natrule_Inv pfm s を用いればよい．ただし，この不変表明には前提条件がついており，不変表明の等式部分を導出するためには，変換規則 nr が変換表 l に含まれること，つまり，

```
MEM (FIRST (\x. natrule_checkLocalAP x sap s) l) l
```

を証明する必要がある．これは，前提条件

```
nattable_srcnatRuleExists nt sap s
```

を nattable_srcnatRuleExists の定義で書き換えて得られる定理

```
let l = nattable_get_natruleList nt s in
  EXISTS (\x. natrule_checkLocalAP x sap s)  l
```

と，リストに関する定理

```
!P l. EXISTS P l ==> MEM (FIRST P l) l
```

を用いて導出することができる．これにより，サブゴール (1) が証明される．

サブゴール (2) の証明

このサブゴールは，接続は存在しないが，物理的に接続可能な状態にある場合である．pfm_filterOut の定義を展開し，簡単化すると，次のようになる．

```
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre1 pfm s /\
  ~(pfm_connetionExists pfm sap dap s) /\
  pfm_isPhysicallyConnectable pfm sap s /\ ... ==>
    (packet_get_srcAddr p
      (pfm_srcnat pfm p (pfm_addConnection sap dap s)) =
     pfm_getIpAddr pfm s)
```

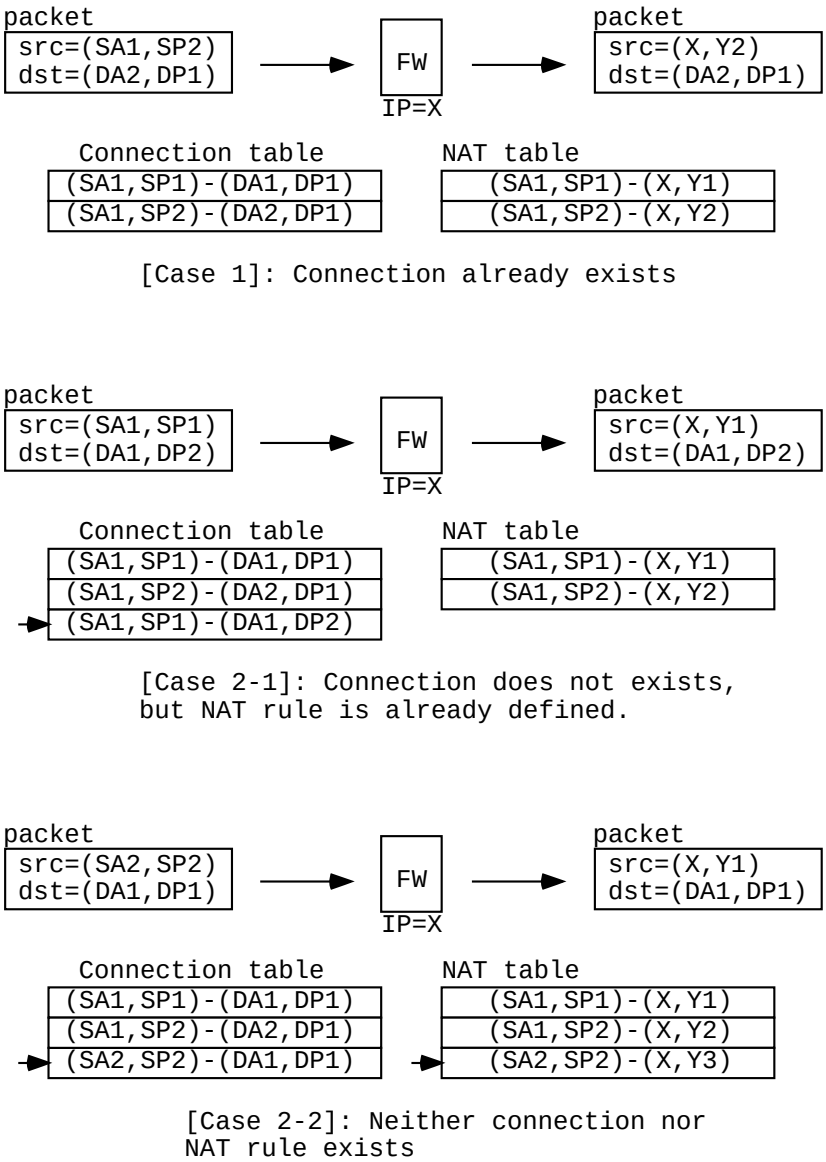

図 9.3: アウトバウンドパケット処理に関する 3 つの場合

ここで，前提条件 pfm_isPhysicallyConnectable pfm sap s を定義により書き換えれば，次の新しい前提条件が得られる．

```
~(pfm_connectionIsFull pfm s) /\
(pfm_srcnatRuleExists pfm sap s \/ ~(pfm_portIsFull pfm s))
```

つまり，物理的に接続可能であることは,「接続表がいっぱいでなく，かつ，送信元アドレス変換規則が存在する，または（変換規則が存在しないときは），未使用ポートが存在する」ことと同値である．この論理和を分解すると，次の 2 つのサブゴールが得られる．

サブゴール (2-1)

```
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  pfm_srcnatRuleExists pfm sap s /\ ... ==>
    (packet_get_srcAddr p
      (pfm_srcnat pfm p (pfm_addConnection sap dap s)) =
     pfm_getIpAddr pfm s)
```

サブゴール (2-2)

```
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  ~(pfm_srcnatRuleExists pfm sap s) /\
  ~(pfm_portIsFull pfm s) /\ ... ==>
    (packet_get_srcAddr p
      (pfm_srcnat pfm p (pfm_addConnection sap dap s)) =
     pfm_getIpAddr pfm s)
```

サブゴール (2-1) の証明

このサブゴールは図 9.3 中段のような，接続は存在しないが送信元アドレス変換規則が既に存在する，という場合である．

ゴールを分解していくとサブゴール (1) の途中と同じ形

```
let sap = packet_getSrcAP p s in
let nt = pfm_get_nattable pfm s in
let l = nattable_get_natruleList nt s in
let nr = FIRST (\x. natrule_checkLocalAP x sap s) l in
  nattable_srcnatRuleExists nt sap s /\ ... ==>
    (FST (natrule_get_globalAP nr s) = pfm_getIpAddr pfm s)
```

が得られる．この後は (1) と同様の手順により証明することができる．

サブゴール (2-2) の証明

(1), (2-1) ともに，pfm_filterOut を適用する前から sap に対応する送信元アドレス変換規則が存在している場合，つまり，最初から

```
nattable_srcnatRuleExists nt sap s
```

が成立している場合である．この事実は，前者の場合,「接続が存在する」という事実と,「接続が存在するならばそれに対応する送信元アドレス変換規則が存在する」という不変表明から導出することができた．後者の場合は，接続は存在しないが,「物理的に接続可能」という条件の中の 1 つである,「送信元アドレス変換規則が存在する」という条件に合致した場合として得ることができた．いずれの場合も，最初から存在する送信元アドレス変換規則によってパケットの送信元アドレスが書き換えられるわけであるが，その書き換えられた値がファイアウォールの公開アドレスと一致することは,「すべての変換規則について変換先 IP アドレスがファイアウォールの公開アドレスである」という不変表明を用いて証明することができた．(2-2) は，図 9.3 下段に示すような，接続も変換規則も存在しない場合であり，pfm_filterOut の途中で動的に変換規則が追加され，それをもとにパケットのアドレス変換が行われる．

ゴールを分解していくと (1) や (2-1) の途中の命題と似た形

```
let sap = packet_getSrcAP p s in
let nt = pfm_get_nattable pfm s in
let s1 = nattable_addNatrule nt sap s in
let l = nattable_get_natruleList nt s1 in
let nr = FIRST (\x. natrule_checkLocalAP x sap s1) l in
  ~(nattable_srcnatRuleExists nt sap s) /\ ... ==>
    (FST (natrule_get_globalAP nr s1) = pfm_getIpAddr pfm s)
```

が得られる．このゴールでは，変換規則のリスト l は nattable_addNatrule が適用された後のリスト，つまり，新しく生成した変換規則 FST (natrule_new s) を追加した後の変換表であり，nr はその追加されたリストにおいて sap に対応する変換規則となっている．nattable_addNatrule により変換表に追加された規則は sap に対応する規則であるから，nr は追加された規則そのものとなっているはずである．これは次のように導出できる．

```
nr = FIRST (\x. natrule_checkLocalAP x sap s1) l
   = FIRST (\x. natrule_checkLocalAP x sap s1)
       (FST (natrule_new s)::(nattable_get_natruleList nt s))
   = if natrule_checkLocalAP (FST (natrule_new s)) sap s1 then
       FST (natrule_new s)
     else FIRST (nattable_get_natruleList nt s)
```

ここで，s1 における新しい変換規則 FST (natrule_new s) の内部アドレス（変換元アドレス）は sap に設定されているため，if の条件は真となる．したがって，nr = FST (natrule_new s) がいえる．

これにより，ゴールは次のように簡単化される．

```
let sap = packet_getSrcAP p s in
let nt = pfm_get_nattable pfm s in
let s1 = nattable_addNatrule nt sap s in
let nr = FST (natrule_new s) in
  ~(nattable_srcnatRuleExists nt sap s) /\ ... ==>
    (FST (natrule_get_globalAP nr s1) = pfm_getIpAddr pfm s)
```

さらに，分解を進めると次のゴールが得られる．

```
let sap = packet_getSrcAP p s in
let nt = pfm_get_nattable pfm s in
let nr = FST (natrule_new s) in
let gap = (pfm_getIpAddr pfm s, nattable_getPort nt s) in
let s1 = natrule_set_globalAP nr gap (SND (natrul_new s)) in
  ~(nattable_srcnatRuleExists nt sap s) /\ ... ==>
    (FST (natrule_get_globalAP nr s1) = pfm_getIpAddr pfm s)
```

ここで，s1 において変換規則 nr の変換先アドレスが gap，つまり，ファイアウォールの公開 IP アドレス pfm_getIpAddr pfm s と動的に得られる未使用ポート nattable_getPort nt s の組に設定されていることが分かる．したがって，s1 において取得される nr の変換先 IP アドレスは当然 pfm_getIpAddr pfm s となる．これにより，等式の左辺と右辺が一致し，サブゴール (2-2) が証明される．

以上で，すべての場合の証明が完了し，求める定理が得られる．

9.3 接続表とアドレス変換表の一貫性証明 (1)

接続表とアドレス変換表の間の一貫性として，「接続表に接続が存在するならば必ずその内部アドレスに対応する送信元アドレス変換規則がアドレス変換表に存在する」という性質を証明する．図 9.4 に示すように，例えば，(SA1,SP2)-(DA2,DP1) という接続が存在するとき，その内部アドレス (SA1,SP2) に対応して，必ず，(SA1,SP2)-(X,Y2) といった送信元アドレス変換規則が存在する．

この不変表明は前の証明で前提条件 con_nat_Inv として用いたように，アドレス変換が正しく行われるためにも必要な不変表明である．

不変表明を証明するためには，「不変表明がシステムの初期状態で成立し，すべての（外側から呼び出される）メソッドの適用前後で成立する」ことを証明しなければならない．接続表，及びアドレス変換表を変更する可能性のある操作は次の 4 つである．

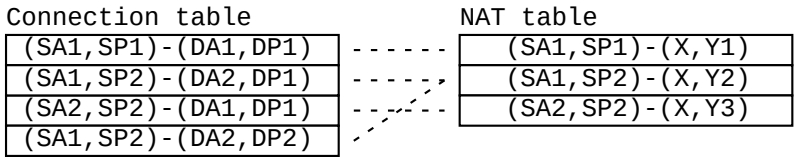

図 9.4: 接続表とアドレス変換表の一貫性

- pfm_stop（パケットフィルタの停止）
- pfm_filterOut（アウトバウンドパケットフィルタ）
- pfm_filterIn（インバウンドパケットフィルタ）
- pfm_incSec（タイムアウト時間の減算）

ここでは pfm_filterOut, pfm_incSec の適用前後で con_nat_Inv が成立することを証明する．まず，本節で前者の証明を行い，次節で後者の証明を行う．

命題の設定

目的の不変表明 con_nat_Inv を証明するためには，接続や変換規則の内部アドレスや外部アドレスだけでなく，タイムアウト残り時間も考慮に入れなければならない．これは特に pfm_incSec が，タイムアウト残り時間に依存して表から要素を削除するからである．したがって，内部アドレスと外部アドレスだけに着目した不変表明 con_nat_Inv は，単独で証明するには不十分であり，タイムアウト時間も含めたより強い不変表明を証明する必要がある．

タイムアウト時間も考慮に入れた不変表明 con_nat_Inv2 を次のように定義する．

```
|- !pfm s. con_nat_Inv2 pfm s =
  con_nat_timer_Inv pfm s /\ connection_timer_Inv pfm s
where
|- !pfm s. con_nat_timer_Inv pfm s =
     !lap gap t.
       pfm_connectionExistsWithTime pfm lap gap t s ==>
       pfm_srcnatRuleExistsWithTime pfm lap t s
|- !pfm s. connection_timer_Inv pfm s =
     let ct = pfm_get_contable pfm s in
     let l = contable_get_connectionList ct s in
       !x. MEM x l ==> 0 < connection_get_timer x s
|- !pfm s. pfm_connectionExistsWithTime pf lap gap t s =
     let ct = pfm_get_contable pfm s in
```

```
     let l = contable_get_connectionList ct s in
       EXISTS (\x. connection_check x lap dap s /\
                t < connection_get_timer x s) l
|- !pfm s. pfm_srcnatRuleExistsWithTime pf lap t s =
     let nt = pfm_get_nattable pfm s in
     let l = nattable_get_natruleList nt s in
       EXISTS (\x. natrule_checkLocalAP x lap s /\
                t < natrule_get_timer x s) l
```

con_nat_Inv2 は，2 つの不変表明 con_nat_timer_Inv と connection_timer_Inv の論理積である．con_nat_timer_Inv は,「内部アドレス lap と外部アドレス gap の間の接続が存在し，タイムアウト残り時間が t より大きいならば，lap に対する送信元アドレス変換規則が存在し，タイムアウト残り時間が t より大きい」を意味する．connection_timer_Inv は,「接続表に含まれるすべての接続について，タイムアウト残り時間が 0 より大きい」を意味する．con_nat_Inv2 は con_nat_Inv を含意する（証明は後述）．

証明対象の命題は次のようになる．

```
let (msg,p1,s1) = pfm_filterOut pfm p s in
Pre2 pfm s /\ con_nat_Inv2 pfm s ==> con_nat_Inv2 pfm s1
```

つまり，ストア s について con_nat_Inv2 が成立することを仮定し，pfm_filterOut の適用後のストア s1 について con_nat_Inv2 が依然として成立していることを証明する．

前提条件 Pre2 は，接続表，アドレス変換表に関する基本的な仮定や不変表明をまとめたものであり，次のように定義される．

```
|- !pfm s. Pre2 pfm s =
     pfm_ex pfm s /\  (1)
     contable_ex_Inv pfm s /\  (2)
     nattable_ex_Inv pfm s /\  (3)
     connection_ex_Inv pfm s /\  (4)
     natrule_ex_Inv pfm s /\  (5)
     connection_count_Inv pfm s /\  (6)
     natrule_count_Inv pfm s /\  (7)
     connection_timer_max_Inv pfm s /\  (8)
     natrule_timer_max_Inv pfm s /\  (9)
     connection_limit_Inv pfm s /\  (10)
     natrule_limit_Inv pfm s /\  (11)
     con_nat_limit_Inv pfm s  (12)
where
  |- !pfm s. connection_count_Inv pfm s =
       let ct = pfm_get_contable pfm s in
```

```
      let l = contable_get_connectionList ct s in
        !x. MEM x l ==> (COUNT x l = 1)
|- !pfm s. natrule_count_Inv pfm s =
      let nt = pfm_get_nattable pfm s in
      let l = nattable_get_natruleList ct s in
        !x. MEM x l ==> (COUNT x l = 1)
|- !pfm s. connection_timer_max_Inv pfm s =
      let ct = pfm_get_contable pfm s in
      let l = contable_get_connectionList ct s in
        !x. MEM x l ==>
          connection_get_timer x s <= pfm_getConnectionTime pfm s
|- !pfm s. natrule_timer_max_Inv pfm s =
      let nt = pfm_get_nattable pfm s in
      let l = nattable_get_natruleList nt s in
        !x. MEM x l ==>
          natrule_get_timer x s <= pfm_getNatruleTime pfm s
|- !pfm s. connection_limit_Inv pfm s =
      (0 < pfm_getConnectionTime pfm s)
|- !pfm s. natrule_limit_Inv pfm s = (0 < pfm_getNatruleTime pfm s)
|- !pfm s. con_nat_limit_Inv pfm s =
      (pfm_getConnectionTime pfm s <= pfm_getNatruleTime pfm s)
```

(1) は pfm オブジェクトがストアに存在するという仮定である．(2)(3) はそれぞれ,「pfm オブジェクトとリンクする接続表オブジェクトは必ずストアに存在する」,「pfm オブジェクトとリンクするアドレス変換表オブジェクトは必ずストアに存在する」を意味する不変表明である．(4)(5) はそれぞれ,「接続表に含まれる接続オブジェクトが必ずストアに存在する」,「アドレス変換表に含まれる変換規則オブジェクトが必ずストアに存在する」を意味する不変表明である．(6)(7) はそれぞれ,「接続表に同じ接続オブジェクトが重複して含まれない」,「アドレス変換表に同じ変換規則オブジェクトが重複して含まれない」を意味する不変表明である．

(8)(9) はそれぞれ,「接続のタイムアウト残り時間はシステムが規定する接続タイムアウト時間以下である」,「アドレス変換規則のタイムアウト残り時間はシステムが規定するアドレス変換規則タイムアウト時間以下である」を意味する不変表明である．接続タイムアウト時間とは，contable クラスの属性 timeLimit の値であり，pfm_setConnectionTime により設定される値である．また，アドレス変換規則タイムアウト時間とは，nattable クラスの属性 timeLimit の値であり，pfm_setNatruleTime により設定される値である．接続，アドレス変換規則のタイムアウト残り時間は，pfm_setConnectionTime, pfm_setNatruleTime によって設定された後は，pfm_incSec により減算されるだけであり，設定値を超えて加算するようなメソッドは存在しない．したがって，この不変表明が成立することは明らか

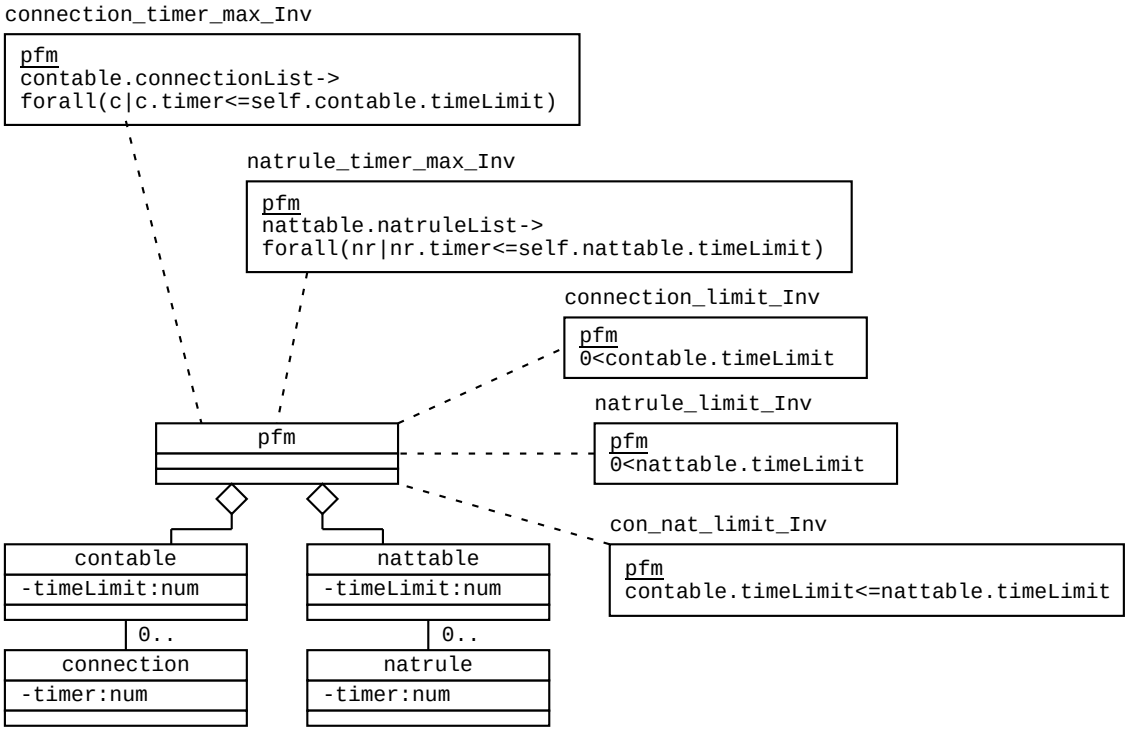

図 9.5: 前提不変表明の OCL による記述

である．

(10)(11) はそれぞれ,「接続タイムアウト時間が正の値である」,「アドレス変換規則タイムアウト時間が正の値である」を意味する不変表明である．接続タイムアウト時間，アドレス変換規則タイムアウト時間は，それぞれ pfm_setConnectionTime, pfm_setConnectionTime によってのみ設定可能であるが，両方とも，正の値しか設定できないようになっている．したがって，この不変表明が成立することは明らかである．

(12) は,「接続タイムアウト時間がアドレス変換規則タイムアウト時間以下である」を意味する不変表明である．pfm_setConnectionTime では，pfm_getNatruleTime により取得される値より大きい値を設定することは不可能となっている．また，pfm_setNatruleTime では，pfm_getConnectionTime により取得される値より小さな値を設定することは不可能となっている．したがって，この不変表明が成立することは明らかである．

(8)〜(12) の不変表明の OCL による記述を図 9.5 に示す．

不変表明（強）から不変表明（弱）の導出

本題の証明に移る前に，con_nat_Inv2 が con_nat_Inv を含意すること，つまり，次の命題が成立することを証明しておく．

```
!pfm s. con_nat_Inv2 pfm s ==> con_nat_Inv pfm s
```

まず，全称量子を除去し，con_nat_Inv2 の定義を展開する．

```
con_nat_timer_Inv pfm s /\ connection_timer_Inv pfm s
==> con_nat_Inv pfm s
```

さらに，con_nat_timer_Inv, con_nat_Inv の定義を展開する．

```
(!lap gap t.
  pfm_connectionExistsWithTime pfm lap gap t s ==>
  pfm_srcnatRuleExistsWithTime pfm lap t s) /\  (1)
connection_timer_Inv pfm s /\
pfm_connectionExists pfm lap gap s ==>
  pfm_srcnatRuleExists pfm lap s
```

ここで，前提条件 (1) の全称量子をそれぞれ，lap, gap, 0 に具体化する．

```
(pfm_connectionExistsWithTime pfm lap gap 0 s ==>
 pfm_srcnatRuleExistsWithTime pfm lap 0 s) /\  (1)
connection_timer_Inv pfm s /\  (2)
pfm_connectionExists pfm lap gap s  (3)
==> pfm_srcnatRuleExists pfm lap s
```

ここで，次の定理を証明することができる．

```
|- connection_timer_Inv pfm s /\
   pfm_connectionExists pfm lap gap s ==>
     pfm_connectionExistsWithTime pfm lap gap 0 s
```

つまり，すべての接続のタイムアウト残り時間が 0 以上であり，かつ，lap, gap に対応する接続が存在するならば，タイムアウト残り時間が 0 以上の lap, gap に対応する接続が存在する．

この定理と，前提条件 (2)(3) より，新しい前提条件

```
pfm_connectionExistsWithTime pfm lap gap 0 s
```

が得られる．

さらに，この前提条件と (1) より，新しい前提条件

```
pfm_srcnatRuleExistsWithTime pfm lap 0 s
```

が得られる．

ここで，次の定理を証明することができる．

```
|- pfm_srcnatRuleExistsWithTime pfm lap 0 s ==>
   pfm_srcnatRuleExists pfm lap s
```

つまり，タイムアウト残り時間が 0 以上の lap に対応する送信元アドレス変換規則が存在するならば，当然，lap に対応する送信元アドレス変換規則が存在する．

この定理を用いてゴールの結論を導出することができる．

証明

本題の証明を行う．証明対象の命題は次のようになる．

```
let (msg,p1,s1) = pfm_filterOut pfm p s in
Pre2 pfm s /\ con_nat_Inv2 pfm s ==> con_nat_Inv2 pfm s1
```

p1 = packet_null の場合は，pfm_filterOut は接続表，アドレス変換表に対し一切変更を加えていないので成立することは明らかである．よって，~(p1 = packet_null) を前提条件に加えた次の命題の証明を行う．

```
let (msg,p1,s1) = pfm_filterOut pfm p s in
Pre2 pfm s /\ ~(p1 = packet_null) /\
  con_nat_Inv2 pfm s ==> con_nat_Inv2 pfm s1
```

con_nat_Inv2 は，2 つの不変表明の論理積であるからそれらを補題 (A)(B) として別々に証明する．

```
(A) !pfm s. let (msg,p1,s1) = pfm_filterOut pfm p s in
      Pre2 pfm s /\ ~(p1 = packet_null) /\
      con_nat_Inv2 pfm s ==> con_nat_timer_Inv pfm s1

(B) !pfm s. let (msg,p1,s1) = pfm_filterOut pfm p s in
      Pre2 pfm s /\ ~(p1 = packet_null) /\
      con_nat_Inv2 pfm s ==> connection_timer_Inv pfm s1
```

以下，補題 (A)(B) を順に証明する．

補題 (A) の証明

証明対象の命題は次の通りである．

```
!pfm s. let (msg,p1,s1) = pfm_filterOut pfm p s in
  Pre2 pfm s /\ ~(p1 = packet_null) /\
  con_nat_Inv2 pfm s ==> con_nat_timer_Inv pfm s1
```

まず，9.1 で証明した次の定理に基づき，ゴールを 2 つのサブゴールに分解する．

サブゴール (1)

```
let (msg,p1,s1) = pfm_filterOut pfm p s in
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre2 pfm s /\
  pfm_connectionExists pfm sap dap s /\
  con_nat_Inv2 pfm s /\ ... ==> con_nat_timer_Inv pfm s1
```

サブゴール (2)

```
let (msg,p1,s1) = pfm_filterOut pfm p s in
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre2 pfm s /\
  ~(pfm_connectionExists pfm sap dap s) /\
  pfm_isPhysicallyConnectable pfm sap s /\
  con_nat_Inv2 pfm s /\ ... ==> con_nat_timer_Inv pfm s1
```

サブゴール (1) の証明

このサブゴールは，パケット p の送信元アドレス sap と宛先アドレス dap の間に接続が既に存在する場合である．この場合，図 9.6 に示すように，接続表に対しては，contable_update により，sap と dap に対応する接続のタイムアウト残り時間が初期化される．また，不変表明の仮定より，sap に対応する送信元アドレス変換規則が存在する（前提条件 con_nat_Inv2 から con_nat_Inv が導出され，これと前提条件 pfm_connectionExists pfm sap dap s から pfm_srcnatRuleExists pfm sap s が導出される）．したがって，アドレス変換表に対しては，新しい変換規則は生成されず，nattable_srcnat1 により，sap に対応する既存の送信元アドレス変換規則が使用され，タイムアウト残り時間が初期化される．

実際に定義を展開し，簡単化すると次のようになる．

```
let ct = pfm_get_contable pfm s in
let nt = pfm_get_nattable pfm s in
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre2 pfm s /\
  con_nat_timer_Inv pfm s /\ ... ==>
    (pfm_connectionExistsWithTime pfm lap gap t
      (contable_update ct sap dap s) ==>
     pfm_srcnatRuleExistsWithTime pfm lap t
```

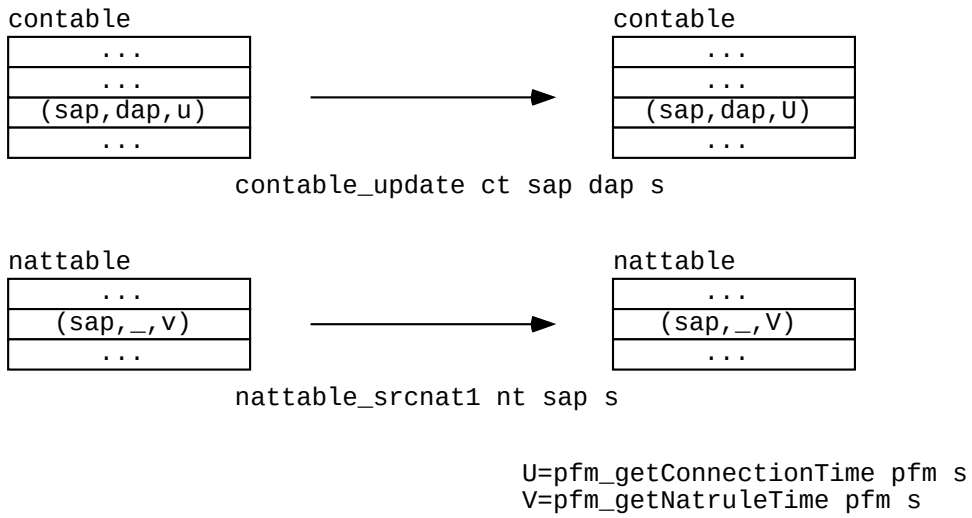

図 9.6: pfm_filterOut による接続表とアドレス変換表の変化 (1)

```
(SND (nattable_srcnat1 pfm sap s)))
```

ゴールに含まれる次の項

```
pfm_connectionExistsWithTime pfm lap gap t (contable_update ct sap dap s)
```

は，前提条件や定義を用いて分解していくと，

```
(lap = sap) /\ (gap = dap) /\ (t < pfm_getConnectionTime pfm s) \/
pfm_connectionExistsWithTime pfm lap gap t s
```

という項に簡単化することができる．つまり，送信元アドレス sap と宛先アドレス dap の間の接続のタイムアウト残り時間を更新した直後に，内部アドレス lap と外部アドレス gap の間に残り時間が t より大きい接続が存在することは，lap と gap がいままさに更新された sap と dap にそれぞれ一致し，その設定された残り時間 pfm_getConnectionTime pfm s が t より大きいか，または，更新前から，(更新された接続以外で) lap と gap の間に残り時間が t より大きい接続が存在していたかのどちらかである．

また，次の項

```
pfm_srcnatRuleExistsWithTime pfm lap t (SND (nattable_srcnat1 pfm sap s))
```

は，前提条件や定義を用いて分解していくと，

```
(lap = sap) /\ (t < pfm_getNatruleTime pfm s) \/
pfm_srcnatRuleExistsWithTime pfm lap t s
```

に簡単化できる．つまり，パケット p に対し送信元アドレス変換を行った直後に，lap に対応する残り時間が t より大きい変換規則が存在することは，lap がいままさに使用された変換規則の送信元アドレス sap に一致し，その設定された残り時間 pfm_getNatruleTime pfm s が t より大きいか，または，変換前から，(使用された変換規則以外で) lap に対応する残り時間が t より大きい接続が存在していたかのどちらかである．

以上により，ゴールは次のように簡単化される．

```
Pre2 pfm s /\
con_nat_timer_Inv pfm s /\ ... ==>
  (lap = sap) /\  (1)
  (gap = dap) /\  (2)
  (t < pfm_getConnectionTime pfm s) \/  (3)
  pfm_connectionExistsWithTime pfm lap gap t s  (4)
  ==> (lap = sap) /\  (5)
      (t < pfm_getNatruleTime pfm s) \/  (6)
      pfm_srcnatRuleExistsWithTime pfm lap t s  (7)
```

(1)(2)(3) が成立する場合，(1) から直接 (5) が導出され，(3) と Pre2 に含まれる不変表明 con_nat_limit_Inv から (6) が導出される．また，(4) が成立する場合，不変表明の仮定 con_nat_timer_Inv より (7) が導出される．

以上により，サブゴール (1) の証明が完了する．

サブゴール (2) の証明

このサブゴールは，接続は存在しないが，物理的に接続可能な状態にある場合である．

前提条件にある pfm_isPhysicallyConnectable pfm sap s を定義により書き換え，その論理和を分解すると，次の 2 つのサブゴールが得られる．

サブゴール (2-1)

```
let (msg,p1,s1) = pfm_filterOut pfm p s in
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre2 pfm s /\
  ~(pfm_connectionExists pfm sap dap s) /\
  pfm_srcnatRuleExists pfm sap s /\
  con_nat_Inv2 pfm s /\ ... ==> con_nat_timer_Inv pfm s1
```

サブゴール (2-2)

```
let (msg,p1,s1) = pfm_filterOut pfm p s in
let sap = packet_getSrcAP p s in
```

```
let dap = packet_getDstAP p s in
  Pre2 pfm s /\
  ~(pfm_connectionExists pfm sap dap s) /\
  ~(pfm_srcnatRuleExists pfm (packet_getSrcAP p s) s) /\
  ~(pfm_portIsFull pfm p s) /\
  con_nat_Inv2 pfm s /\ ... ==> con_nat_timer_Inv pfm s1
```

サブゴール (2-1) の証明

このサブゴールは，接続は存在しないが，送信元アドレス変換規則が既に存在する場合である．この場合，図 9.7 に示すように，接続表に対しては contable_addConnection により，sap と dap の間の接続が追加される．また，アドレス変換表に対しては，新しい変換規則は生成されず，nattable_srcnat1 により，sap に対応する既存の送信元アドレス変換規則が使用され，タイムアウト残り時間が初期化される．

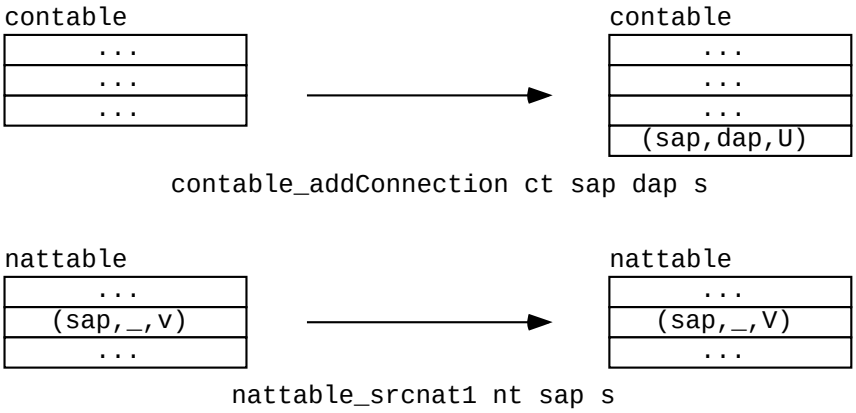

図 9.7: pfm_filterOut による接続表とアドレス変換表の変化 (2)

実際に定義を展開し，簡単化すると次のようになる．

```
let ct = pfm_get_contable pfm s in
let nt = pfm_get_nattable pfm s in
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre2 pfm s /\
  con_nat_timer_Inv pfm s /\ ... ==>
    (pfm_connectionExistsWithTime pfm lap gap t
       (contable_addConnection ct sap dap s) ==>
     pfm_srcnatRuleExistsWithTime pfm lap t
       (SND (nattable_srcnat1 nt sap s))
```

ゴールに含まれる次の項

```
pfm_connectionExistsWithTime pfm lap gap t (pfm_addConnection pfm sap dap s)
```

は，前提条件や定義を用いて分解していくと，

```
(lap = sap) /\ (gap = dap) \/ (t < pfm_getConnectionTime pfm s) \/
pfm_connectionExistsWithTime pfm lap gap t s
```

という項に簡単化することができる．つまり，送信元アドレス sap と宛先アドレス dap の間に接続を追加した直後に，内部アドレス lap と外部アドレス gap の間に残り時間が t より大きい接続が存在することは，lap と gap がいままさに追加された sap と dap にそれぞれ一致し，その設定された残り時間 pfm_getConnectionTime pfm s が t より大きいか，または，接続を追加する前から lap と gap の間に残り時間が t より大きい接続が存在していたかのどちらかである．

また，次の項

```
pfm_srcnatRuleExistsWithTime pfm lap t (SND (contable_srcnat1 ct sap s))
```

は，サブゴール (1) の証明と同様に，

```
(lap = sap) /\ (t < pfm_getNatruleTime pfm s) \/
pfm_srcnatRuleExistsWithTime pfm lap t s
```

に簡単化できる．

この後の証明は，サブゴール (1) の証明と同様である．

サブゴール (2-2) の証明

このサブゴールは，接続も送信元アドレス変換規則も存在しない場合である．この場合，図 9.8 に示すように，接続表に対しては contable_addConnection により，sap と dap の間の接続が追加される．また，アドレス変換表に対しては，nattable_addNatRule により，sap に対応する変換規則が生成され，さらに，nattable_srcnat1 により，生成した変換規則が使用され，タイムアウト残り時間が初期化される．

実際に定義を展開し，簡単化すると次のようになる．

```
let ct = pfm_get_contable pfm s in
let nt = pfm_get_nattable pfm s in
let sap = packet_getSrcAP p s in
let dap = packet_getDstAP p s in
  Pre2 pfm s /\
  (!lap gap t.
   pfm_connectionExistsWithTime pfm lap gap t s ==>
```

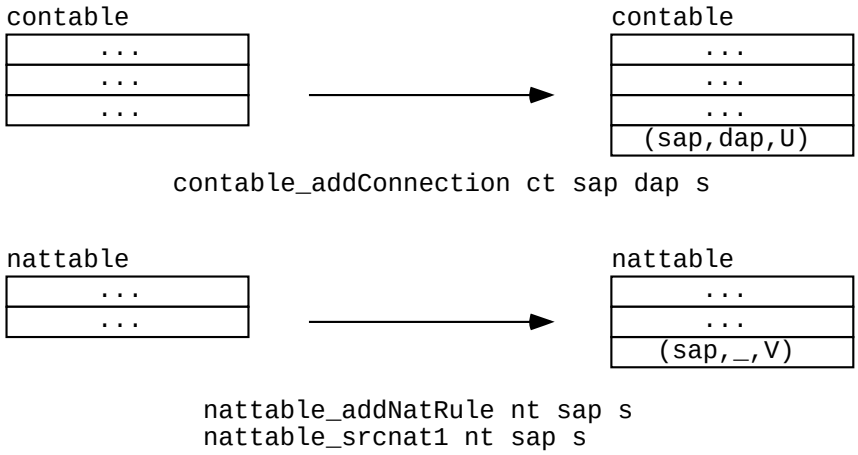

図 9.8: pfm_filterOut による接続表とアドレス変換表の変化 (3)

```
pfm_srcnatRuleExistsWithTime pfm lap t s) /\ ... ==>
 (pfm_connectionExistsWithTime pfm lap gap t
   (contable_addConnection ct sap dap s) ==>
  pfm_srcnatRuleExistsWithTime pfm lap t
    (SND (nattable_srcnat1 pfm sap
      (nattable_addNatRule nt sap s)))
```

次の項

```
pfm_connectionExistsWithTime pfm lap gap t
  (pfm_addConnection pfm sap dap s)
```

は，サブゴール (2-1) の証明と同様に，

```
(lap = sap) /\ (gap = dap) /\ (t < pfm_getConnectionTime pfm s) \/
pfm_connectionExistsWithTime pfm lap gap t s
```

と簡単化できる．

また，次の項

```
pfm_srcnatRuleExistsWithTime pfm lap t
  (SND (nattable_srcnat1 pfm sap (nattable_addNatRule nt sap s)))
```

は，

```
(lap = sap) /\ (t < pfm_getNatruleTime pfm s) \/
pfm_srcnatRuleExistsWithTime pfm lap t s
```

と簡単化できる．つまり，パケット p の送信元アドレス sap に対応する変換規則を生成し，その変換規則により変換を行った直後，lap に対応する残り時間が t より大きい変換規則が存在することは，lap がいままさに追加された変換規則の送信元アドレス sap に一致し，その設定された残り時間 pfm_getNatruleTime pfm s が t より大きいか，または，変換前から lap に対応する残り時間が t より大きい接続が存在していたかのどちらかである．

この後の証明は，サブゴール (1) の証明と同様である．

以上ですべてのサブゴールの証明が完了し，求める定理が得られる．

補題 (B) の証明

証明対象の命題は次の通りである．

```
!pfm s. let (msg,p1,s1) = pfm_filterOut pfm p s in
  Pre2 pfm s /\ ~(p1 = packet_null) /\
  con_nat_Inv2 pfm s ==> connection_timer_Inv pfm s1
```

pfm_filterOut においては接続の生成，または更新の際に，接続のタイムアウト時間が設定される．設定される値は，pfm_getConnectionTime によって取得される値であるため，不変表明の仮定 connection_limit_Inv により正の値であることが保証されている．したがって，pfm_filterOut 適用後も，すべての接続のタイムアウト残り時間は正の値となる（詳細略）．

9.4 接続表とアドレス変換表の一貫性証明 (2)

pfm_incSec の適用前後で不変表明 con_nat_Inv2 が成立することを証明する．

命題の設定

pfm_incSec は，pfm_incSec1 を n 回繰り返すメソッドであるため，実際には，pfm_incSec1 の適用前後で不変表明が成立することを証明すれば，pfm_incSec の適用前後でも成立することがいえる．

証明対象の命題は次のようになる．

```
let s1 = pfm_incSec1 pfm s in
  Pre2 pfm s /\ con_nat_Inv2 pfm s ==> con_nat_Inv2 pfm s1
```

つまり，ストア s について con_nat_Inv2 が成立することを仮定し，pfm_incSec1 の適用後のストア s1 について con_nat_Inv2 が依然として成立していることを証明する．

証明

9.3 と同様，次の 2 つの補題を証明する．

```
(A) !pfm s. Pre2 pfm s /\ con_nat_timer_Inv pfm s ==>
      con_nat_timer_Inv pfm (pfm_incSec1 pfm s)

(B) !pfm s. Pre2 pfm s ==>
      connection_timer_Inv pfm (pfm_incSec1 pfm s)
```

以下，補題 (A)(B) を順に証明する．

補題 (A) の証明

証明対象の命題は次の通りである．

```
!pfm s. Pre2 pfm s /\ con_nat_timer_Inv pfm s ==>
   con_nat_timer_Inv pfm (pfm_incSec1 pfm s)
```

まず，全称量子を除去し，con_nat_timer_Inv の定義を展開する．

```
Pre2 pfm s /\
(!lap gap t.
  pfm_connectionExistsWithTime pfm lap gap t s ==>
  pfm_srcnatRuleExistsWithTime pfm lap t s) ==>
 (pfm_connectionExistsWithTime pfm lap gap t (pfm_incSec1 pfm s) ==>
  pfm_srcnatRuleExistsWithTime pfm lap t (pfm_incSec1 pfm s))
```

ここで，次の定理が成立することに着目する．

```
|- !pfm s. Pre2 pfm s ==>
   (pfm_connectionExistsWithTime pfm lap gap t (pfm_incSec1 pfm s) =
    pfm_connectionExistsWithTime pfm lap gap (SUC t))
```

つまり，すべての接続についてタイムアウト時間を 1 減算した直後に，lap と gap の間の残り時間が t となっている接続が存在することは，タイムアウト減算前に lap と gap の間の残り時間が SUC t となっている接続が存在することと同値である（図 9.9 上段）．この定理は，定義を展開後，接続表を表す次の項

```
contable_get_connectionList (pfm_get_contable pfm s) s
```

に関する帰納法を用いて証明することができる（証明略）．

アドレス変換表についても同様に次の定理が成立する．

```
|- !pfm s. Pre2 pfm s ==>
   (pfm_srcnatRuleExistsWithTime pfm lap t (pfm_incSec1 pfm s) =
    pfm_srcnatRuleExistsWithTime pfm lap (SUC t))
```

つまり，すべてのアドレス変換規則についてタイムアウト時間を 1 減算した直後に，lap に対応する残り時間が t となっている変換規則が存在することは，タイムアウト減算前に lap に対応する残り時間が SUC t となっている変換規則が存在することと同値である（図 9.9 下段）．

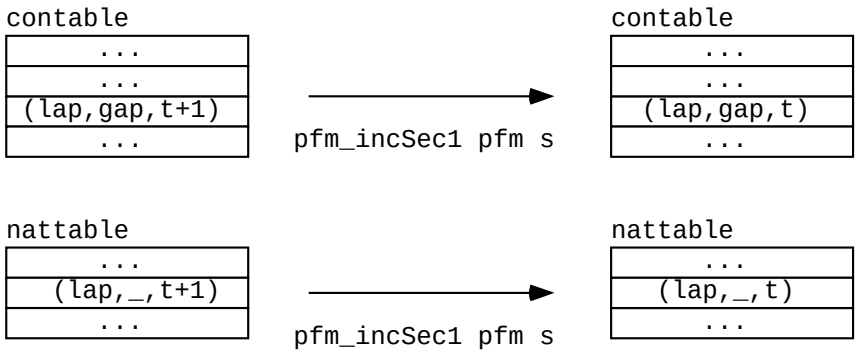

図 9.9: pfm_incSec1 による接続表とアドレス変換表の変化

これらの定理を用いてゴールを簡単化すると，次のようになる．

```
Pre2 pfm s /\
(!lap gap t.
  pfm_connectionExistsWithTime pfm lap gap t s ==>
  pfm_srcnatRuleExistsWithTime pfm lap t s) ==>
 (pfm_connectionExistsWithTime pfm lap gap (SUC t) s ==>
  pfm_srcnatRuleExistsWithTime pfm lap (SUC t) s)
```

後は，前提条件の全称量子をそれぞれ lap, gap, SUC t に特化すれば証明が完了する．

補題 (B) の証明

証明対象の命題は次の通りである．

```
|- !pfm s. Pre2 pfm s ==>
     connection_timer_Inv pfm (pfm_incSec1 pfm s)
```

不変表明 connection_timer_Inv は，pfm_incSec1 適用後，適用前の不変表明の仮定なしで成立する．これが成立する理由は，pfm_incSec1 により，すべての接続のタイムアウト残り時間は 1 減算されるが，減算した結果が 0 となる接続は同時に除去されてしま

うからである．したがって，pfm_incSec1 の適用後は，残り時間が 0 より大きい接続しか残っておらず，不変表明が無条件に成立する（詳細略）．

第III部

付録集

付 録A　パケットフィルタリングシステムの HOLコード

パケットフィルタリングシステムのクラスモデル，HOL におけるメソッド定義を以下に示す．

A.1　クラスモデル

```
class pfm

attr active : bool | false
attr contable : contable | contable_null
attr nattable : nattable | nattable_null
attr frule : frule | frule_null
attr doscounter : doscounter | doscounter_null
attr am : am  | am_null

class packet

attr srcAddr : num | 0
attr dstAddr : num | 0
attr srcPort : num | 0
attr dstPort : num | 0
attr protocol : num | 0

class contable

attr timeLimit : num | 0
attr maxSize : num | 0
attr connectionList : connection list | []

class connection
```

```
attr localAP : num*num | (0,0)
attr globalAP : num*num | (0,0)
attr state : num | 0
attr timer : num | 0

class nattable

attr ipAddr : num | 0
attr ports : (nu,*bool)list | []
attr timeLimit : num | 0
attr natruleList : natrule list | []

class natrule

attr localAP : num*num | (0,0)
attr globalAP : num*num | (0,0)
attr timer : num | 0

class frule

attr srcAddrTable : num list | []
attr dstAddrTable : num list | []
attr srcPortTable : num list | []
attr dstPortTable : num list | []
attr protocolTable : num list | []

class doscounter

attr alert : bool | false
attr count : num | 0
attr threshould : num | 0
```

A.2 定数，ユーティリティ関数の定義

```
(* ヘッダタイプ *)
val SRCADDR = Define `SRCADDR = 0`;;
val DSTADDR = Define `DSTADDR = 1`;;
val SRCPORT = Define `SRCPORT = 2`;;
```

```
val DSTPORT = Define `DSTPORT = 3`;;
val PROTOCOL = Define `PROTOCOL = 4`;;

(* プロトコルタイプ *)
val TCP = Define `TCP = 0`:
val UDP = Define `UDP = 1`;

(* n がプライベートアドレスであるかをチェックする．
   自然数 0..99 が予約範囲とする．*)
val isPrivate = Define
    `isPrivate n = n < 100`;;

(* リスト l の要素で最初に P を満たす要素を取得する．*)
val FIRST = Define
    `FIRST P l = HD (FILTER P l)`;;

(* MONO x n は，すべての要素が x である長さ n のリストである．*)
val MONO = Define
    `(MONO x 0 = []) /\ (MONO x (SUC n) = x::(MONO x n))`;;

(* 連想リストからの要素の取得．
   第 1 要素が x に一致するペアの第 2 要素を取得する．*)
val ASSOC = Define
    `ASSOC x ((y,z)::l) = if x = y then z else ASSOC x l`;;

(* 逆連想リストからの要素の取得．
   第 2 要素が x に一致するペアの第 1 要素を取得する．*)
val REV_ASSOC = Define
    `REV_ASSOC x ((y,z)::l) = if x = z then y else REV_ASSOC x l`;;

(* 連想リストに対する要素の更新．
   第 1 要素が x に一致するペアの第 2 要素を y に更新する．*)
val SET_ASSOC = Define
    `(SET_ASSOC x y [] = []) /\
    (SET_ASSOC x y ((a,b)::l) =
     if x = a then (a,y)::(SET_ASSOC x y l)
     else (a,b)::(SET_ASSOC x y l))`;;
```

A.3 packet クラスのメソッド

- +packet(sa:num,sp:num,da:num,dp:num,pr:num):void

  コンストラクタ．入力は順に，送信元 IP アドレス，ポート番号，宛先 IP アドレス，ポート番号，プロトコルである．

```
val new_packet = Define
    `new_packet sa sp da dp pr s =
    let (p,s) = packet_new s in
    let s = packet_set_srcAddr p sa s in
    let s = packet_set_dstAddr p da s in
    let s = packet_set_srcPort p sp s in
    let s = packet_set_dstPort p dp s in
    let s = packet_set_protocol p pr s in
      (p,s)`;;
```

- +getSrcAP():num*num

  送信元 IP アドレスとポート番号の組を取得する．

```
val packet_getSrcAP = Define
    `packet_getSrcAP p s =
    let sa = packet_get_srcAddr p s in
    let sp = packet_get_srcPort p s in
      (sa,sp)`;;
```

- +setSrcAP(x:num*num):void

  送信元 IP アドレスとポート番号の組を設定する．

```
val packet_setSrcAP = Define
    `packet_setSrcAP p (sa,sp) s =
    let s = packet_set_srcAddr p sa s in
      packet_set_srcPort p sp s`;;
```

- +getDstAP():num*num

  宛先 IP アドレスとポート番号の組を取得する．

```
val packet_getDstAP = Define
    `packet_getDstAP p s =
```

```
    let da = packet_get_dstAddr p s in
    let dp = packet_get_dstPort p s in
      (da,dp)`;;
```

- `+setDstAP(x:num*num):void`

  宛先 IP アドレスとポート番号の組を設定する．

```
val packet_setDstAP = Define
    `packet_setDstAP p (da,dp) s =
    let s1 = packet_set_dstAddr p da s in
      packet_set_dstPort p dp s1`;;
```

- `+getInfo():(num*num)*(num*num)*num`

  パケットのヘッダ情報を取得する．出力は，送信元アドレス，宛先アドレス，プロトコルの組である．

```
val packet_getInfo = Define
    `packet_getInfo p s =
    let sap = packet_getSrcAP p s in
    let dap = packet_getDstAP p s in
    let pr = packet_get_protocol p s in
      (sap,dap,pr)`;;
```

A.4 natrule クラスのメソッド

- `+natrule(lap:num*num,gap:num*num):void`

  コンストラクタ．`lap` は内部アドレス，`gap` は外部アドレスである．

```
val new_natrule = Define
    `new_natrule lap gap s =
    let (nr,s1) = natrule_new s in
    let s = natrule_set_localAP nr lap s in
    let s = natrule_set_globalAP nr gap s in
      (nr,s)`;;
```

- `+checkLocalAP(x:num*num):bool`

  内部アドレスが `x` に一致するかどうかをチェックする．

```
val natrule_checkLocalAP = Define
    `natrule_checkLocalAP nr m s =
    let n = natrule_get_localAP nr s in
      m = n`;;
```

- +checkGlobalAP(x:num*num):bool

  外部アドレスが x に一致するかどうかをチェックする．

```
val natrule_checkGlobalAP = Define
    `natrule_checkGlobalAP nr m s =
    let n = natrule_get_globalAP nr s in
      m = n`;;
```

- +decTimer():bool

  タイムアウト残り時間を 1 減らす．0 となれば true を返す．

```
val natrule_decTimer = Define
    `natrule_decTimer nr s =
    let x = natrule_get_timer nr s - 1 in
    let s1 = natrule_set_timer nr x s in
      (x = 0, s1)`;;
```

- +setTimer(x:num):bool

  タイムアウト残り時間を設定する．

```
val natrule_setTimer = Define
    `natrule_setTimer nr x s = natrule_set_timer nr x s`;;
```

- +getInfo():(num*num)*(num*num)*num

  アドレス変換規則の情報を取得する．出力は，内部アドレス，外部アドレス，タイムアウト時間の組である．

```
val natrule_getInfo = Define
    `natrule_getInfo nr s =
    let lap = natrule_get_localAP nr s in
    let gap = natrule_get_globalAP nr s in
    let t = natrule_get_timer nr s in
      (lap,gap,t)`;;
```

A.5 nattable クラスのメソッド

- +nattable():void

  コンストラクタ.

```
val new_nattable = Define `new_nattable s = nattable_new s`;;
```

- -setIpAddr(x:num):void

  公開 IP アドレスを設定する.

```
val nattable_setIpAddr = Define
    `nattable_setIpAddr nt x s = nattable_set_ipAddr nt x s`;;
```

- -getPort():num

  未使用のポート番号を返すとともに，そのポート番号を使用中にする.

```
val nattable_getPort = Define
    `nattable_getPort nt s =
    let l1 = nattable_get_ports nt s in
    let m = REV_ASSOC T l1 in
    let l2 = SET_ASSOC m F l1 in
    let s = nattable_set_ports nt l2 s in
      (m, s)`;;
```

- -setPorts(l:num list):void

  公開ポート番号のリストを l **に設定する．設定後はすべて未使用状態とする.**

```
val nattable_setPorts = Define
    `nattable_setPorts nt l1 s =
    let l2 = MONO T (LENGTH l1) in
    let l3 = ZIP(l1,l2) in
      nattable_set_ports nt l3 s`;;
```

- -getTimeLimit():num

  変換規則のタイムアウト時間を取得する.

```
val nattable_getTimeLimit = Define
    `nattable_getTimeLimit nt s = nattable_get_timeLimit nt s`;;
```

- -setTimeLimit(x:num):bool

  変換規則のタイムアウト時間を x に設定する．x は正の値でなければならない．

```
val nattable_setTimeLimit = Define
    `nattable_setTimeLimit nt x s =
    if 0 < x then
      let s = nattable_set_timeLimit nt x s in
        (T, s)
    else
      (F, s)`;;
```

- -releasePort(x:num):void

  使用中のポート番号 x を解放する．

```
val nattable_releasePort = Define
    `nattable_releasePort nt m s =
    let l1 = nattable_get_ports nt s in
    let l2 = SET_ASSOC m T l1 in
      nattable_set_ports nt l2 s`;;
```

- +isFull():bool

  ポートがすべて使用中であるかどうかをチェックする．

```
val nattable_isFull = Define
    `nattable_isFull nt s =
    let l1 = nattable_get_ports nt s in
    let l2 = SND (UNZIP l1) in
      EVERY (\x. ~x) l2`;;
```

- +setTimer(nr:natrule):void

  変換規則 nr のタイムアウト時間を初期化する．設定する値は属性 timeLimit の値である．

```
val nattable_setTimer = Define
    `nattable_setTimer nt nr s =
    let x = nattable_get_timeLimit nt s in
      natrule_setTimer nr x s`;;
```

- -addNatrule(lap:num*num):void

  入力内部アドレス lap に対し，アドレス変換規則を生成し，変換表に追加する．変換先の IP アドレスは公開 IP アドレス ipAddr であり，ポート番号は getPort() により得られる値である．

```
val nattable_addNatrule1 = Define
    `nattable_addNatrule1 nt nr s =
    let l = nattable_get_natruleList nt s in
      nattable_set_natruleList nt (nr::l) s`;;

val nattable_addNatrule = Define
    `nattable_addNatrule nt lap s =
    let a = nattable_get_ipAddr nt s in
    let (p,s) = nattable_getPort nt s in
    let (nr,s) = new_natrule lap (a,p) s in
    let s = nattable_setTimer nt nr s in
      nattable_addNatrule1 nt nr s`;;
```

- +srcnatRuleExists(sap:num*num):bool

  入力アドレス sap に対する．送信元アドレス変換規則が存在するかどうかをチェックする．

```
val nattable_srcnatRuleExists = Define
    `nattable_srcnatRuleExists nt sap s =
    let l = nattable_get_natruleList nt s in
      EXISTS (\x. natrule_checkLocalAP x sap s) l`;;
```

- +dstnatRuleExists(dap:num*num):bool

  入力アドレス dap に対する宛先アドレス変換規則が存在するかどうかをチェックする．

```
val nattable_dstnatRuleExists = Define
    `nattable_dstnatRuleExists nt dap s =
    let l = nattable_get_natruleList nt s in
      EXISTS (\x. natrule_checkGlobalAP x dap s) l`;;
```

- +srcnat(sap:num*num):num*num

  入力アドレス sap の送信元アドレス変換を行う．出力は変換後のアドレスである．送信元アドレス変換規則が存在しない場合は，変換表に空きがあるときに限り，新しい変換規則を生成し，その規則により変換する．空きがないときは，sap をそのまま出力する．変換を行った場合はその変換規則のタイムアウト時間を初期化する．

```
val nattable_srcnat1 = Define
    `nattable_srcnat1 nt sap s =
    let l = nattable_get_natruleList nt s in
    let nr = FIRST (\x. natrule_checkLocalAP x sap s) l in
    let sap1 = natrule_get_globalAP nr s in
    let s = nattable_setTimer nt nr s in
      (sap1,s)`;;

val nattable_srcnat = Define
    `nattable_srcnat nt sap s =
    if nattable_srcnatRuleExists nt sap s then
      nattable_srcnat1 nt sap s
    else if ~(nattable_isFull nt s) then
      let s = nattable_addNatrule nt sap s in
        nattable_srcnat1 nt sap s
    else
      (sap,s)`;;
```

- `+dstnat(dap:num*num):num*num`

  入力アドレス `dap` **の宛先アドレス変換を行う．出力は変換後のアドレスである．宛先アドレス変換規則が存在しない場合は，** `dap` **をそのまま出力する．変換を行った場合はその変換規則のタイムアウト時間を初期化する．**

```
val nattable_dstnat1 = Define
    `nattable_dstnat1 nt dap s =
    let l = nattable_get_natruleList nt s in
    let nr = FIRST (\x. natrule_checkGlobalAP x dap s) l in
    let dap1 = natrule_get_localAP nr s in
    let s = nattable_setTimer nt nr s in
      (dap1,s)`;;

val nattable_dstnat = Define
    `nattable_dstnat nt dap s =
    if nattable_dstnatRuleExists nt dap s then
      nattable_dstnat1 nt dap s
    else
      (dap,s)`;;
```

- `+decTimer():num`

すべてのアドレス変換規則について，そのタイムアウト残り時間を 1 減らすとともに，0 となった規則を消去する．

```
val nattable_decTimer1 = Define
    `(nattable_decTimer1 (nt:nattable) [] l s = (l,s)) /\
    (nattable_decTimer1 nt (nr::l1) l2 s =
     let (b,s) = natrule_decTimer nr s in
       if b then
         let gap = natrule_get_globalAP nr s in
         let s2 = nattable_releasePort nt (SND gap) s in
           nattable_decTimer1 nt l1 l2 s
       else
         nattable_decTimer1 nt l1 (nr::l2) s)`;;

val nattable_decTimer = Define
    `nattable_decTimer nt s =
    let l1 = nattable_get_natruleList nt s in
    let (l2,s) = nattable_decTimer1 nt l1 [] s in
      nattable_set_natruleList nt l2 s`;;
```

- +reset():num

  アドレス変換規則リストを空にし，すべてのポートを未使用状態にする．

```
val nattable_reset = Define
    `nattable_reset nt s =
    let l = nattable_get_ports nt s in
    let l1 = FST (UNZIP l) in
    let l2 = MONO T (LENGTH l1) in
    let l3 = ZIP(l1,l2) in
    let s = nattable_set_ports nt l3 s in
      nattable_set_natruleList nt [] s`;;
```

- +getNatrules():((num*num)*(num*num)*num)list

  すべてのアドレス変換規則の情報をリストにして表示する．

```
val nattable_getNatrules = Define
    `nattable_getNatrules nt s =
    let l = nattable_get_natruleList nt s in
      MAP (\x. natrule_getInfo x s) l`;;
```

A.6 frule クラスのメソッド

- `+frule():void`

  コンストラクタ.

```
val new_frule = Define `new_frule s = frule_new s`;;
```

- `-checkSrcAddr(x:num):bool`

  `x` **が通過可能な送信元** IP **アドレスであるかどうかをチェックする.**

```
val frule_checkSrcAddr = Define
    `frule_checkSrcAddr fr x s = MEM x (frule_get_srcAddrTable fr s)`;;
```

- `-checkSrcAddr(x:num):bool`

  `x` **が通過可能な宛先** IP **アドレスであるかどうかをチェックする.**

```
val frule_checkDstAddr = Define
    `frule_checkDstAddr fr x s = MEM x (frule_get_dstAddrTable fr s)`;;
```

- `-checkSrcPort(x:num):bool`

  `x` **が通過可能な送信元ポート番号であるかどうかをチェックする.**

```
val frule_checkSrcPort = Define
    `frule_checkSrcPort fr x s = MEM x (frule_get_srcPortTable fr s)`;;
```

- `-checkDstPort(x:num):bool`

  `x` **が通過可能な宛先ポート番号であるかどうかをチェックする.**

```
val frule_checkDstPort = Define
    `frule_checkDstPort fr x s = MEM x (frule_get_dstPortTable fr s)`;;
```

- `-checkProtocol(x:num):bool`

  `x` **が通過可能なプロトコルタイプであるかどうかをチェックする.**

```
val frule_checkProtocol = Define
    `frule_checkProtocol fr x s = MEM x (frule_get_protocolTable fr s)`;;
```

- +check(p:packet):bool

  p が通過可能なパケットであるかどうかをチェックする．

```
val frule_check = Define
    `frule_check fr p s =
    let sa = packet_get_srcAddr p s in
    let da = packet_get_dstAddr p s in
    let sp = packet_get_srcPort p s in
    let dp = packet_get_dstPort p s in
    let pr = packet_get_protocol p s in
      frule_checkSrcAddr fr sa s /\ frule_checkDstAddr fr da s /\
      frule_checkSrcPort fr sp s /\ frule_checkDstPort fr dp s /\
      frule_checkProtocol fr pr s`;;
```

- +getFilterRules(ty:num):num list

  ヘッダタイプ ty に対応するフィルタルールを取得する．ty は，SRCADDR, DSTADDR, SRCPORT, DSTPORT, PROTOCOL のいずれかである．

```
val frule_getFilterRules = Define
    `frule_getFilterRules fr ty s =
    if ty = SRCADDR then frule_get_srcAddrTable fr s
    else if ty = DSTADDR then frule_get_dstAddrTable fr s
    else if ty = SRCPORT then frule_get_srcPortTable fr s
    else if ty = DSTPORT then frule_get_dstPortTable fr s
    else if ty = PROTOCOL then frule_get_protocolTable fr s
    else []`;;
```

- +getFilterRules(ty:num,l:num list):void

  ヘッダタイプ ty に対応するフィルタルールを設定する．

```
val frule_setFilterRules = Define
    `frule_setFilterRules fr ty l s =
    if ty = SRCADDR then frule_set_srcAddrTable fr l s
    else if ty = DSTADDR then frule_set_dstAddrTable fr l s
    else if ty = SRCPORT then frule_set_srcPortTable fr l s
    else if ty = DSTPORT then frule_set_dstPortTable fr l s
    else if ty = PROTOCOL then frule_set_protocolTable fr l s
    else s`;;
```

A.7 doscounter クラスのメソッド

- +doscounter():void

  コンストラクタ.

```
val new_doscounter = Define `new_doscounter s = doscounter_new s`;;
```

- +getThreshould():num

  カウンタの閾値を取得する.

```
val doscounter_getThreshould = Define
    `doscounter_getThreshould dc s = doscounter_get_threshould dc s`;;
```

- +setThreshould(x:num):bool

  カウンタの閾値を x **に設定する.** x **は正の値でなければならない.**

```
val doscounter_setThreshould = Define
    `doscounter_setThreshould dc x s =
    if 0 < x then
      let s = doscounter_set_threshould dc x s in
        (T, s)
    else
      (F, s)`;;
```

- +inc():void

  カウンタを 1 **増加する. 閾値に達したとき,** DoS **警告フラグを** ON **にする.**

```
val doscounter_inc = Define
    `doscounter_inc dc s =
    let x = doscounter_get_count dc s + 1 in
    let s = doscounter_set_count dc x s in
    let y = doscounter_get_threshould dc s in
    if y <= x then
      doscounter_set_alert dc T s
    else
      s`;;
```

- +reset():void

  カウンタをリセットする.

```
val doscounter_reset = Define
    `doscounter_reset dc s = doscounter_set_count dc 0 s`;;
```

- `+getAlert():void`

  DoS 警告フラグを取得する．

```
val doscounter_getAlert = Define
    `doscounter_getAlert dc s = doscounter_get_alert dc s`;;
```

- `+resetAlert():void`

  DoS 警告フラグを OFF にする．

```
val doscounter_resetAlert = Define
    `doscounter_resetAlert dc s = doscounter_set_alert dc F s`;;
```

A.8 connection クラスのメソッド

- `+connection(lap:num*num,gap:num*num):void`

  コンストラクタ．`lap` は内部アドレス，`gap` は外部アドレスである．

```
val new_connection = Define
    `new_connection lap gap s =
    let (c,s) = connection_new s in
    let s = connection_set_localAP c lap s in
    let s = connection_set_globalAP c gap s in
      (c,s)`;;
```

- `+check(lap:num*num,gap:num*num):bool`

  内部アドレス `lap`，外部アドレス `gap` の接続が存在するかどうかをチェックする．

```
val connection_check = Define
    `connection_check c lap gap s =
    let lap1 = connection_get_localAP c s in
    let gap1 = connection_get_globalAP c s in
      (lap1 = lap) /\ (gap1 = gap)`;;
```

- `+setTimer(x:num):void`

  タイムアウト残り時間を `x` に設定する．

```
val connection_setTimer = Define
    `connection_setTimer c x s = connection_set_timer c x s`;;
```

- +decTimer():bool

  タイムアウト残り時間を 1 減らす．0 となれば true を返す．

```
val connection_decTimer = Define
    `connection_decTimer c s =
    let x = connection_get_timer c s - 1 in
    let s = connection_set_timer c x s in
      (x = 0, s)`;;
```

- +getInfo():(num*num)*(num*num)*num

  接続情報を取得する．出力は，内部アドレス，外部アドレス，タイムアウト時間の組である．

```
val connection_getInfo = Define
    `connection_getInfo c s =
    let lap = connection_get_localAP c s in
    let gap = connection_get_globalAP c s in
    let t = connection_get_timer c s in
      (lap,gap,t)`;;
```

A.9 contable クラスのメソッド

- +contable():void

  コンストラクタ．

```
val new_contable = Define `new_contable s = contable_new s`;;
```

- -getMaxSize():num

  接続の最大数を取得する．

```
val contable_getMaxSize = Define
    `contable_getMaxSize x s = contable_get_maxSize x s`;;
```

- -setMaxSize(x:num):bool

  接続の最大数を x に設定する．

```
val contable_setMaxSize = Define
    `contable_setMaxSize ct x s = contable_set_maxSize ct x s`;;
```

- -getTimeLimit():num

  接続のタイムアウト時間を取得する．

```
val contable_getTimeLimit = Define
    `contable_getTimeLimit x s = contable_get_timeLimit x s`;;
```

- -setTimeLimit(x:num):bool

  接続のタイムアウト時間を x **に設定する．** x **は正の値でなければならない．**

```
val contable_setTimeLimit = Define
    `contable_setTimeLimit ct x s =
    if 0 < x then
      let s = contable_set_timeLimit ct x s in
        (T, s)
      else
        (F, s)`;;
```

- -setTimer(c:connection):void

  入力される接続 c **のタイムアウト時間を初期化する．設定する値は属性** timeLimit **の値である．**

```
val contable_setTimer = Define
    `contable_setTimer ct c s =
    let x = contable_get_timeLimit ct s in
      connection_setTimer c x s`;;
```

- +isFull():bool

  接続が最大数に達しているかどうかをチェックする．

```
val contable_isFull = Define
    `contable_isFull ct s =
    let x = contable_get_maxSize ct s in
    let l = contable_get_connectionList ct s in
      x = LENGTH l`;;
```

- -getConnection(lap:num*num,gap:num*num):connection

  内部アドレス lap**，外部アドレス** gap **に対応する接続を取得する．**

```
val contable_getConnection = Define
    `contable_getConnection ct lap gap s =
    let l = contable_get_connectionList ct s in
      FIRST (\c. connection_check c lap gap s) l`;;
```

- +connectionExists(lap:num*num,gap:num*num):bool

  内部アドレス lap**，外部アドレス** gap **に対応する接続が存在するかどうかをチェックする．**

```
val contable_connectionExists = Define
    `contable_connectionExists ct lap gap s =
    let l = contable_get_connectionList ct s in
      EXISTS (\c. connection_check c lap gap s) l`;;
```

- +addConnection(lap:num*num,gap:num*num):void

  内部アドレスが lap**，外部アドレスが** gap **の新しい接続を追加する．同時に，追加した接続表のタイムアウト時間を初期化する．接続表がいっぱいであれば何も行わない．**

```
val contable_addConnection1 = Define
    `contable_addConnection1 ct c s =
    let l = contable_get_connectionList ct s in
      contable_set_connectionList ct (c::l) s`;;

val contable_addConnection = Define
    `contable_addConnection ct lap gap s =
    if ~(contable_isFull ct s) then
      let l = contable_get_connectionList ct s in
      let (c,s) = new_connection lap gap s in
      let s = contable_addConnection1 ct c s in
        contable_setTimer ct c s
    else
      s`;;
```

- +decTimer():void

  すべての接続について，そのタイムアウト残り時間を 1 **減らすとともに，**0 **となった接続を削除する．**

```
val contable_decTimer1 = Define
    `(contable_decTimer1 (ct:contable) [] l s = (l,s)) /\
    (contable_decTimer1 ct (c::l1) l2 s =
    let (b,s) = connection_decTimer c s in
      if b then
        contable_decTimer1 ct l1 l2 s
      else
        contable_decTimer1 ct l1 (c::l2) s)`;;

val contable_decTimer = Define
    `contable_decTimer ct s =
    let l1 = contable_get_connectionList ct s in
    let (l2,s) = contable_decTimer1 ct l1 [] s in
      contable_set_connectionList ct l2 s`;;
```

- `+reset():void`

  接続リストを空にする．

```
val contable_reset = Define
    `contable_reset ct s =
    contable_set_connectionList ct [] s`;;
```

- `+update(lap:num*num,gap:num*num):void`

  内部アドレス `lap`**，外部アドレス** `gap` **の接続のタイムアウト時間を初期化する．**

```
val contable_update = Define
    `contable_update ct lap gap s =
    let c = contable_getConnection ct lap gap s in
      contable_setTimer ct c s`;;
```

- `+getConnections():((num*num)*(num*num)*num)list`

  すべての接続情報をリストにして表示する．

```
val contable_getConnections = Define
    `contable_getConnections ct s =
    let l = contable_get_connectionList ct s in
      MAP (\x. connection_getInfo x s) l`;;
```

A.10 pfm クラスのメソッド

- +pfm():void

  コンストラクタ．4 つの集約オブジェクトを生成し，リンクを張る．

```
val new_pfm = Define
    `new_pfm s =
    let (pfm,s) = pfm_new s in
    let (dc,s) = new_doscounter s in
    let s = pfm_set_doscounter pfm dc s in
    let (fr,s) = new_frule s in
    let s = pfm_set_frule pfm fr s in
    let (nt,s) = new_nattable s in
    let s = pfm_set_nattable pfm nt s in
    let (ct,s) = new_contable s in
    let s = pfm_set_contable pfm ct s in
      (pfm, s)`;;
```

- -isActive():bool

  パケットフィルタリング機能が作動中であるかどうかをチェックする．

```
val pfm_isActive = Define
    `pfm_isActive pfm s = pfm_get_active pfm s`;;
```

- -getMaxConnectionSize():num

  接続の最大数を取得する．

```
val pfm_getMaxConnectionSize = Define
    `pfm_getMaxConnectionSize pfm s =
    let ct = pfm_get_contable pfm s in
      contable_get_maxSize ct s`;;
```

- -setMaxConnectionSize(x:num):bool

  接続の最大数を x に設定する．パケットフィルタが停止時のみ設定可能．

```
val pfm_setMaxConnectionSize = Define
    `pfm_setMaxConnectionSize pfm x s =
      if pfm_isActive pfm s then
```

```
        (F, s)
      else
        let ct = pfm_get_contable pfm s in
        let s = contable_setMaxSize ct x s in
          (T, s)`;;
```

- -getConnetionTime():num

  接続のタイムアウト時間を取得する．

```
val pfm_getConnectionTime = Define
    `pfm_getConnectionTime pfm s =
    let ct = pfm_get_contable pfm s in
      contable_getTimeLimit ct s`;;
```

- -setConnectionTime(x:num):bool

  接続のタイムアウト時間を x **に設定する．パケットフィルタが停止時のみ設定可能であり，**x **は正の値でなければならない．また，**x **はアドレス変換規則のタイムアウト時間の値を上回ってはならない．これは，接続が存在するならば必ずそれに対応する変換規則が存在することを保証するためである．**

```
val pfm_setConnectionTime = Define
    `pfm_setConnectionTime pfm x s =
    let ct = pfm_get_contable pfm s in
    let y = pfm_getNatruleTime pfm s in
      if ~pfm_isActive pfm s /\ 0 < x /\ x <= y then
        contable_setTimeLimit ct x s
      else
        (F, s)`;;
```

- -getNatruleTime():num

  アドレス変換規則のタイムアウト時間を取得する．

```
val pfm_getNatruleTime = Define
    `pfm_getNatruleTime pfm s =
    let nt = pfm_get_nattable pfm s in
      nattable_getTimeLimit nt s`;;
```

- -setNatruleTime(x:num):bool

  アドレス変換規則のタイムアウト時間を x に設定する．id は設定を行うユーザ ID，x は設定する値である．パケットフィルタが停止時のみ設定可能であり，x は正の値でなければならない．また，x は接続のタイムアウト時間の値を下回ってはならない．これは，接続が存在するならば必ずそれに対応する変換規則が存在することを保証するためである．

```
val pfm_setNatruleTime = Define
    `pfm_setNatruleTime pfm x s =
    let nt = pfm_get_nattable pfm s in
    let y = pfm_getConnectionTime pfm s in
      if ~pfm_isActive pfm s /\ 0 < x /\ y <= x then
        nattable_setTimeLimit nt x s
      else
        (F, s)`;;
```

- +getIpAddr():num

  公開 IP アドレスを取得する．

```
val pfm_getIpAddr = Define
    `pfm_getIpAddr pfm s =
    let nt = pfm_get_nattable pfm s in
      nattable_get_ipAddr nt s`;;
```

- +setIpAddr(x:num):bool

  公開 IP アドレスを設定する．パケットフィルタ停止時のみ設定可能である．

```
val pfm_setIpAddr = Define
    `pfm_setIpAddr pfm x s =
    if pfm_isActive pfm s then
      (F, s)
    else
      let nt = pfm_get_nattable pfm s in
      let s = nattable_setIpAddr nt x s in
        (T, s)`;;
```

- +getPorts():num list

  公開ポート番号のリストを取得する．

```
val pfm_getPorts = Define
    `pfm_getPorts pfm s =
    let nt = pfm_get_nattable pfm s in
      nattable_get_ports nt s`;;
```

- +setPorts(l:num list):void

  公開ポート番号のリストを設定する．パケットフィルタ停止時のみ設定可能である．

```
val pfm_setPorts = Define
    `pfm_setPorts pfm l s =
    if pfm_isActive pfm s then
      (F, s)
    else
      let nt = pfm_get_nattable pfm s in
      let s = nattable_setPorts nt l s in
        (T, s)`;;
```

- -reset():bool

  接続表，アドレス変換表，ポート使用状態を初期化する．

```
val pfm_reset = Define
    `pfm_reset pfm s =
    let nt = pfm_get_nattable pfm s in
    let ct = pfm_get_contable pfm s in
    let dc = pfm_get_doscounter pfm s in
    let s = doscounter_reset dc s in
    let s = nattable_reset nt s in
      contable_reset ct s`;;
```

- +start():bool

  パケットフィルタの起動．停止時のみ起動可能である．

```
val pfm_start = Define
    `pfm_start pfm s =
    if pfm_isActive pfm s then
      (F, s)
    else
      let s = pfm_set_active pfm T s in
        (T, s)`;;
```

- +stop():bool

  パケットフィルタの停止．同時に，接続表，アドレス変換表，ポート使用状態のリセットを行う．作動時のみ停止可能である．

```
val pfm_stop = Define
    `pfm_stop pfm s =
    if ~pfm_isActive pfm s then
      (F,s)
    else
      let s = pfm_reset pfm s in
      let s = pfm_set_active pfm F s in
        (T,s)`;;
```

- -isOutbound(p:packet):bool

  入力パケットがアウトバウンドであるかどうかをチェックする．具体的には，送信元アドレスがプライベートアドレスであり，宛先アドレスが公開アドレスであり，かつ，ファイアウォールの公開アドレスでないことをチェックする．

```
val pfm_isOutbound = Define
    `pfm_isOutbound pfm p s =
    let x = pfm_getIpAddr pfm s in
    let sa = packet_get_srcAddr p s in
    let da = packet_get_dstAddr p s in
       isPrivate sa /\ ~(isPrivate da) /\ ~(da = x)`;;
```

- -isValidPacket(p:packet):bool

  パケット p のヘッダ情報が有効であるかどうかをチェックする．具体的には，p がアウトバウンドであり，かつ，フィルタルールを満たすことをチェックする．

```
val pfm_isValidPacket = Define
    `pfm_isValidPacket pfm p s =
    let fr = pfm_get_frule pfm s in
      pfm_isOutbound pfm p s /\ frule_check fr p s`;;
```

- -srcnat(p:packet):void

  パケット p の送信元アドレス変換を行う．nattable クラスのメソッド srcnat() により変換後の送信アドレスを取得し，p の送信元アドレスを書き換える．

```
val pfm_srcnat = Define
    `pfm_srcnat pfm p s =
    let sap1 = packet_getSrcAP p s in
    let nt = pfm_get_nattable pfm s in
    let (sap2,s) = nattable_srcnat nt sap1 s in
      packet_setSrcAP p sap2 s`;;
```

- -dstnat(p:packet):void

  パケット p の宛先アドレス変換を行う．nattable クラスのメソッド dstnat() により変換後の宛先アドレスを取得し，p の宛先アドレスを書き換える．

```
val pfm_dstnat = Define
    `pfm_dstnat pfm p s =
    let dap1 = packet_getDstAP p s in
    let nt = pfm_get_nattable pfm s in
    let (dap2,s) = nattable_dstnat nt dap1 s in
      packet_setDstAP p dap2 s`;;
```

- -drop(p:packet)

  パケット p の通過が拒否された場合に行う処理．DoS カウンタの増加を行う．

```
val pfm_drop = Define
    `pfm_drop pfm s =
    let dc = pfm_get_doscounter pfm s in
      doscounter_inc dc s`;;
```

- -srcnatRuleExists(sap:num*num):bool

  入力アドレス sap に対する送信元アドレス変換規則が定義されているかどうかをチェックする．

```
val pfm_srcnatRuleExists = Define
    `pfm_srcnatRuleExists pfm sap s =
    let nt = pfm_get_nattable pfm s in
      nattable_srcnatRuleExists nt sap s`;;
```

- -dstnatRuleExists(dap:num*num):bool

  入力アドレス dap に対する宛先アドレス変換規則が定義されているかどうかをチェックする．

```
val pfm_dstnatRuleExists = Define
    `pfm_dstnatRuleExists pfm dap s =
    let nt = pfm_get_nattable pfm s in
      nattable_dstnatRuleExists nt dap s`;;
```

- -connectionExists(lap:num*num,gap:num*num):bool

  内部アドレス lap **と外部アドレス** gap **の間に接続が存在するかどうかをチェックする．**

```
val pfm_connectionExists = Define
    `pfm_connectionExists pfm lap gap s =
    let ct = pfm_get_contable pfm s in
      contable_connectionExists ct lap gap s`;;
```

- -connectionIsFull():bool

  接続がいっぱいであるかどうかをチェックする．

```
val pfm_connectionIsFull = Define
    `pfm_connectionIsFull pfm s =
    let ct = pfm_get_contable pfm s in
      contable_isFull ct s`;;

val pfm_portIsFull = Define
    `pfm_portIsFull pfm s =
    let nt = pfm_get_nattable pfm s in
      nattable_isFull nt s`;;
```

- -addConnection(lap:num*num,gap:num*num):void

  内部アドレスが lap**，外部アドレスが** gap **の新しい接続を追加する．**

```
val pfm_addConnection = Define
    `pfm_addConnection pfm lap gap s =
    let ct = pfm_get_contable pfm s in
      contable_addConnection ct lap gap s`;;
```

- -updateConnection(lap:num*num,gap:num*num):void

  内部アドレス lap**，外部アドレス** gap **の接続について，そのタイムアウト時間を初期化する．既存の接続に属すパケットが通過した際に呼び出される．**

```
val pfm_updateConnection = Define
    `pfm_updateConnection pfm lap gap s =
    let ct = pfm_get_contable pfm s in
      contable_update ct lap gap s`;;
```

- `-isPhysicallyConnectable(sap:num*num):void`

  入力内部アドレスについて新しい接続を生成することが物理的に可能であるかどうかをチェックする．物理的に接続可能であるとは，接続表がいっぱいでなく，未使用ポートが存在することである．ただし，未使用ポートがない場合でも，入力アドレスに対応する送信元変換規則がすでに存在している場合は接続可能である．

```
val pfm_isPhysicallyConnectable = Define
    `pfm_isPhysicallyConnectable pfm sap s =
    ~pfm_connectionIsFull pfm s /\
    (pfm_srcnatRuleExists pfm sap s \/ ~pfm_portIsFull pfm s)`;;
```

- `+filterOut(p:packet):string*packet`

  アウトバウンド送信のパケット `p` **に対してパケットフィルタリングを行う．出力は結果を表すメッセージと送信元アドレス変換後のパケットの組である．拒否された場合は** `packet_null` **を返す．**

```
val pfm_filterOut = Define
    `pfm_filterOut pfm p s =
    if ~pfm_isActive pfm s then
      ("drop: pf not active", packet_null, s)
    else
      let sap = packet_getSrcAP p s in
      let dap = packet_getDstAP p s in
        if pfm_connectionExists pfm sap dap s then
          let s = pfm_srcnat pfm p s in
          let s = pfm_updateConnection pfm sap dap s in
            ("pass: existing connection", p, s)
        else if pfm_isPhysicallyConnectable pfm sap s then
      if pfm_isValidPacket pfm p s then
        let s = pfm_addConnection pfm sap dap s in
        let s = pfm_srcnat pfm p s in
          ("pass: new connection", p, s)
          else
```

```
            let s = pfm_drop pfm s in
              ("drop: rule not match", packet_null, s)
        else
          let s = pfm_drop pfm s in
            ("drop: physically unconnectable", packet_null, s)`;;
```

- +filterIn(p:packet):string*packet

  インバウンド送信のパケット p **に対してパケットフィルタリングを行う．出力は結果を表すメッセージと宛先アドレス変換後のパケットの組である．拒否された場合は** packet_null **を返す．**

```
val pfm_filterIn = Define
    `pfm_filterIn pfm p s =
    if ~pfm_isActive pfm s then
      ("drop: pf not active", packet_null, s)
    else
      let dap = packet_getDstAP p s in
        if ~pfm_dstnatRuleExists pfm dap s then
          let s = pfm_drop pfm s in
            ("drop: no dstnat rule", packet_null, s)
        else
          let s = pfm_dstnat pfm p s in
          let sap = packet_getSrcAP p s in
          let dap = packet_getDstAP p s in
            if pfm_connectionExists pfm dap sap s then
              let s = pfm_updateConnection pfm dap sap s in
                ("pass: existing connection", p, s)
            else
              let s = pfm_drop pfm s in
                ("drop: inbound not permitted", packet_null, s)`;;
```

- +getFilterRules(ty:num):num list

  ヘッダタイプ ty **に対応するフィルタルールを取得する．**

```
val pfm_getFilterRules = Define
    `pfm_getFilterRules pfm ty s =
    let fr = pfm_get_frule pfm s in
      frule_getFilterRules fr ty s`;;
```

- +setFilterRules(ty:num,l:num list):bool

  ヘッダタイプ ty に対応するフィルタルールを l に設定する．パケットフィルタが停止時のみ設定可能である．

```
val pfm_setFilterRules = Define
    `pfm_setFilterRules pfm ty l s =
    if pfm_isActive pfm s then
      (F,s)
    else
      let fr = pfm_get_frule pfm s in
      let s = frule_setFilterRules fr ty l s in
        (T,s)`;;
```

- +getDosThreshould():num

  DoS カウンタの閾値の取得．

```
val pfm_getDosThreshould = Define
    `pfm_getDosThreshould pfm s =
    let dc = pfm_get_doscounter pfm s in
      doscounter_getThreshould dc s`;;
```

- +setDosThreshould(x:num):bool

  DoS カウンタの閾値を x に設定する．パケットフィルタが停止時のみ設定可能である．

```
val pfm_setDosThreshould = Define
    `pfm_setDosThreshould pfm x s =
    if pfm_isActive pfm s then
      (F, s)
    else
      let dc = pfm_get_doscounter pfm s in
        doscounter_setThreshould dc x s`;;
```

- +getNatrules():((num*num)*(num*num)*num)list

  すべてのアドレス変換規則の情報をリストにして表示する．

```
val pfm_getNatrules = Define
    `pfm_getNatrules pfm s =
    let nt = pfm_get_nattable pfm s in
      nattable_getNatrules nt s`;;
```

- +getConnections():((num*num)*(num*num)*num)list

  すべての接続情報をリストにして表示する．

```
val pfm_getConnections = Define
    `pfm_getConnections pfm s =
    let nt = pfm_get_contable pfm s in
      contable_getConnections nt s`;;
```

- +getAlert():void

  DoS **警告フラグを取得する．**

```
val pfm_getDosAlert = Define
    `pfm_getDosAlert pfm s =
    let dc = pfm_get_doscounter pfm s in
      doscounter_getAlert dc s`;;
```

- +resetAlert():void

  DoS **警告フラグを** OFF **にする．**

```
val pfm_resetDosAlert = Define
    `pfm_resetDosAlert pfm s =
    let dc = pfm_get_doscounter pfm s in
      doscounter_resetAlert dc s`;;
```

- +incSec(n:num):void

  時間を n **秒進める．**1 **秒ごとに，**DoS **カウンタのリセット，接続，アドレス変換規則のタイムアウト残り時間の減算を行う．**

```
val pfm_incSec1 = Define
    `pfm_incSec1 pfm s =
    let ct = pfm_get_contable pfm s in
    let nt = pfm_get_nattable pfm s in
    let dc = pfm_get_doscounter pfm s in
    let s = contable_decTimer ct s in
    let s = nattable_decTimer nt s in
      doscounter_reset dc s`;;

val pfm_incSec = Define
```

```
`(pfm_incSec pfm 0 s = s) /\
(pfm_incSec pfm (SUC n) s =
 let s = pfm_incSec1 pfm s in
  pfm_incSec pfm n s)`;;
```

付 録B　公理・演算子対応表

オブジェクト指向理論の公理と演算子の対応関係を以下に示す．

Axioms	*New*	*Ex*	*Get*	*Set*	*Cast*	*Is*	*Null*	*Emp*
NOT_EX_EMP		○						○
NOT_EX_NULL		○					○	
EX_IS		○				○		
NOT_EX_FST_NEW	○	○						
NOT_EX_FST_NEW_CAST	○	○			○			
IS_IMP_NOT_IS						○		
IS_CAST					○	○		
IS_NEW	○					○		
IS_NEW_CAST	○				○	○		
DIFF_IS_NEW	○					○		
IS_SET				○		○		
DOWN_NULL					○	○	○	
NOT_EX_CAST		○			○		○	
UP_11		○			○			
DOWN_11					○	○		
UP_DOWN		○			○			
DOWN_UP		○			○			
CAST_CAST					○			
EX_CAST_NEW	○	○			○			
DIFF_CAST_NEW	○				○			

表 B.1: **公理・演算子対応表** (1)

Axioms	*New*	*Ex*	*Get*	*Set*	*Cast*	*Is*	*Null*	*Unknown*
NOT_EX_GET		○	○					○
NOT_EX_SET		○		○				
SPR_GET			○		○			
SPR_SET				○	○			
GET_SET		○	○	○				
DIFF_OBJ_GET_SET			○	○				
DIFF_GET_SET			○	○				
GET_NEW	○		○					
GET_NEW_CAST	○		○		○			
EX_GET_NEW	○	○	○					
DIFF_GET_NEW	○		○					
SET_SET				○				
DIFF_OBJ_SET_SET				○				
DIFF_SET_SET				○				
EX_SET_NEW	○	○		○				
DIFF_SET_NEW	○			○				
DIFF_NEW_NEW	○							
SET_GET			○	○				
FST_NEW_SET	○			○				
DIFF_FST_NEW_NEW	○							

表 B.2: **公理・演算子対応表** (2)

付 録C　HOLにおける論理記号の表記

本書で使われる HOL の論理記号を以下に示す．

HOL における表記	標準的な表記	意味
T	$\top$	真
F	$\bot$	偽
~	$\neg$	否定
/\	$\wedge$	論理積
\/	$\vee$	論理和
==>	$\Rightarrow$	含意
!	$\forall$	全称
?	$\exists$	存在
\	λ	抽象
#	$*$	直積
FST	Fst	直積の第 1 要素
SND	Snd	直積の第 2 要素

表 C.1: 論理記号の表記

関連図書

[1] OMG. Unified Modeling Language.
URL: http://www.omg.org/.

[2] The HOL system.
URL: http://hol.sourceforge.net/.

[3] Moscow ML.
URL: http://www.dina.dk/ sestoft/mosml.html.

[4] J. Warmer and A. Kleppe. The object constraint language: precise modeling with UML. Addison-Wesley, 1999.

[5] M. Abadi and L.Cardeli. A theory of primitive objects. Untyped and first-order systems. Information and Computation, 125(2):78-102, 1996.

[6] Tobias Nipkow, David von Oheimb and Cornelia Pusch. μJava: Embedding a Programming Language in a Theorem Prover. In Foundations of Secure Computation. IOS Press, 2000.

[7] Bart Jacobs et al. LOOP project, http://www.cs.kun.nl/ bart/LOOP/

[8] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. Techn. Rep. CSI-R9924, Comput. Sci. Inst., Univ. of Nijmegen, 1999.

[9] David von Oheimb. Hoare Logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience, vol.13 pp.1173-1214, 2001.

[10] Claude Marché and Christine Paulin-Mohring. Reasoning on Java programs with aliasing and frame conditions. In 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005), LNCS, August 2005.

[11] W. Naraschewski and M. Wenzel. Object-Oriented Verification based on Record Subtyping in Higher-Order Logic. Tecnische Universitat Munchen, 1998.

[12] T.Schafer, A.Knapp, and S.Merz. Model Checking UML State Machines and Collaborations. On the Workshop on Software Model Checking, July 2001. Electronic Notes in Theoretical Computer Science vol.55, n.3.

[13] Using B formal specifications for analysis and verfication of UML/OCL models. Marcano, R. and N. Levy. Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language. Dresden, Germany, October 2002.

[14] K. Lano, D. Clark and K. Androutsopoulos. UML to B: Formal Verification of Object-Oriented Models. Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004.

[15] The Z notation.
URL: http://www.zuser.org/z/.

[16] Kenro Yatake, Toshiaki Aoki and Takuya Katayama. Implementing application-specific Object-Oriented theories in HOL. International Conference on Theoretical Aspects of Computing, ICTAC 2005.

[17] 矢竹健朗．定理証明器 HOL におけるオブジェクト指向理論の構築. 北陸先端科学技術大学院大学，学位論文，2006.

[18] David W. Chapman, Andy Fox 責任編集，糸川洋 訳，シスコシステムズ 監修．Cisco PIX Firewall 実装ガイド．ソフトバンクパブリッシング，2003．

[19] Greg Holden 著，SITE J1 訳．ファイアウォールとネットワークセキュリティ入門—侵入検知と VPN—. トムソンラーニング，2005．