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

Title	高階述語論理定理証明器HOLのための問題領域ライブラリの構築
Author(s)	矢竹, 健朗
Citation	
Issue Date	2002-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1539
Rights	
Description	Supervisor:片山 卓也, 情報科学研究科, 修士

JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

高階述語論理定理証明器HOLのための
問題領域ライブラリの構築

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

矢竹 健朗

2002年3月

修 士 論 文

高階述語論理定理証明器HOLのための問題領域ライブラリの構築

指導教官　片山卓也 教授

審査委員主査　片山卓也 教授
審査委員　権藤克彦 助教授
審査委員　二木厚吉

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

010118 矢竹 健朗

提出年月: 2002 年 2 月

Copyright © 2002 by Yatake Kenrou

目 次

第 1 章　はじめに **1**

第 2 章　分析モデルと領域の理論 **4**

2.1　領域の捉え方 4
2.2　クラスモデル 5
2.3　領域の理論 6
2.3.1　領域の型 6
2.3.2　領域のオペレータ 6
2.3.3　公理 9

第 3 章　定理モジュールの構成 **13**

3.1　定理モジュール 13
3.2　定理モジュール構成の概要 14
3.3　obj1 15
3.3.1　定義 15
3.3.2　定理 16
3.4　obj2 17
3.4.1　定義 17
3.4.2　定理 19
3.5　obj_ax 20
3.5.1　定義 20
3.5.2　定理 23

第 4 章　例題 **29**

4.1　定理モジュール構築の手順 29
4.2　領域分析 29
4.2.1　クラスモデル 29
4.2.2　関数の定義 30
4.2.3　関数の制約記述 36
4.3　HOL への変換と定理証明 37
4.3.1　user_ax,item_ax,srv_ax 37
4.3.2　user 39
4.3.3　item 41
4.3.4　srv 42
4.3.5　貸出システムの定理 45

4.4 型の生成 47
4.4.1 サーバ型の生成 48
4.4.2 利用者型の生成 52

第 5 章 考察 **54**
5.1 証明例 54
5.2 帰納法の回避 57
5.3 命題の設定 60
5.4 NULL ポインタ 60
5.5 定理モジュールの再利用性 61

第 6 章 今後の課題とまとめ **62**
6.1 今後の課題 62
6.2 まとめ 62

謝辞 **64**

参考文献 **65**

第1章　はじめに

背景

オブジェクト指向開発法は大規模システムの開発に非常に有効であり、実際のシステム開発でも採り入れられつつある。

オブジェクト指向開発法の上流工程では、分析モデルが構築される。この分析モデルでは、対象とする概念が非常に抽象的なものであるため、曖昧になりがちであり、それゆえ、矛盾や誤りを含みやすいという問題がある。この誤りが開発の下流工程で発見された場合、修復には非常に大きなコストがかかってしまう。また、発見されない場合はシステムに大きな欠陥をもたらすことになる。

この問題の解決法として、定理証明技術の分析工程への応用手法が提案されている。この手法では、ＵＭＬで提案されている複数のモデルを形式化し、統合写像と呼ばれる概念により、それらを一つのモデルに統合する。統合したモデルから公理系を自動生成し、証明すべき命題を定理証明器ＨＯＬで証明することにより、複数のモデル間の一貫性を保証する。

問題点

現状は、ＨＯＬでの証明は開発者にとって負担が大きい。これは、ＨＯＬには、自然数、リスト、文字列といった原始的な理論しか定義されていないことが原因である。式の中に原始的な要素が多く含まれる場合、その式がアプリケーションレベルでどういった意味を持つのかが理解しにくく、証明の方向性がつかみにくい。また、原始的であればそれだけ事実の積み重ねが足りないということであり、証明には非常に多くのステップを要してしまう。ＨＯＬでの証明を効率化することは、開発を円滑に進める上で極めて重要である。

目的

これを解決する方法として, 現実世界のある問題領域に特化した, 高級なデータ型, オペレータ, 定理を含む定理モジュールを準備しておくというものがある. この高級定理モジュールを用いれば, 証明ステップが削減されるとともに, 式の意味の理解が容易になり, 開発者のもつ問題領域に関する知識を生かした直感的な証明が行えるようになる. 式の意味を理解しながら証明を進めることは, その問題領域の本質を理解することにもつながる.

本研究では, あらゆる領域に適用可能な定理モジュールの構築法を提案する. この方法では, あらゆる領域に適用可能とするために, オブジェクト指向分析モデルを基礎として定理モジュールを構築する. また, オブジェクト指向分析モデルは人間の思考の自然な表現であり, このモデルから定理モジュールを生成することにより, 思考の単位となるような定理を得ることが可能である.

研究内容

まず, あらゆる領域を扱うためのオブジェクト指向の枠組みを HOL に実装した.

- 領域の考え方

  領域を協調動作するオブジェクトの集合と考える. 領域の状態は, 個々のオブジェクトの属性と, オブジェクト間に張られているリンクによって表現する. オブジェクトは関数を持ち, 属性の変更, リンクの張替えを行うことができる. あるオブジェクトの関数呼出しを起点に, 複数のオブジェクトが関数を呼び出しあい, 領域の状態を変化させる.

- クラスモデル

  本研究で導入するオブジェクト指向分析モデルは, 単純にするためにクラスモデルのみと する. クラスは, 属性, リンク集合, 関数を持つ. リンク集合はその値がオブジェクトの集合である特別な属性である.

- 領域の理論

  クラスモデルの意味付けはモデルを領域の理論に対応付けることによって行う. 公理系は, 次の要素から構成される.

  - 領域の型
  - 領域の初期値を表す定数
  - 基本関数の集合
  - オブジェクト生成関数の集合
  - 存在検査述語の集合
  - 複合関数の集合
  - 公理の集合

  基本関数は, 属性操作関数とリンク集合操作関数に分類される. 属性操作関数はモデルにおける属性に対応して与えられる関数であり, 属性の取得, 更新の機能を持つ. リンク集合操作関数はモデルにおけるリンク集合に対応して与えられる関数であり, リンク集合の取得, 初期化, リンクの追加, 削除の機能を持つ. オブジェクト生成関数, 存在検査述語はモデルにおけるクラスに対応して与えられる関数, 述語であり, オブジェクト生成関数は領域内にオブジェクトを生成する機能を持ち, 存在検査述語はオブジェクトが領域内に存在するかどうかを検査する機能を持つ. 複合関数はモデルにおける関数に対応して与えられる関数であり, その式は関数の意味記述をもとに定義される. 公理はこれらのオペレータに関する関係を示すものである.

- HOL での実装

  領域の公理系をＨＯＬの既存の理論から生成する. まず, 一般的なクラスのオペレータの定義と定理を格納した定理モジュール obj_ax を用意した. これはリストに対する要素の追加と削除を基礎として構築される. 特定の領域が与えられたとき, まず, クラスごとに obj_ax を継承した定理モジュールを生成する. 各定理モジュールにおいて obj_ax のオペレータをそのクラスの情報をもとに詳細化し, それらの間の関係を証明する. 得られた各クラスの定理モジュールに対し, それを継承する一つの定理モジュールを生成し, 異なるクラスのオペレータの関係を証明する.

定理モジュールの構築は以下の手順で行う.

1. オブジェクト指向分析モデルによる領域分析
    (a) 領域分析によりクラスモデルを生成する.
    (b) クラスモデルが決定すると関数を定義するための基本関数, オブジェクト生成関数が与えられる. これを用いて関数を定義する. これは模式言語を利用して行う.
    (c) モデルに制約を与える. これは, 関数に事前条件, 事後条件を与えることによって行う. 条件は参照関数から構成される論理式である.
2. モデルを HOL に実装した公理系に変換する.
3. 模式言語で記述された関数定義を HOL の関数定義に変換する.
4. 制約記述を命題に変換し, 生成された公理系の上で証明する.
5. 他のシステムから利用できるような型を生成する.

例題として, 貸出システムについて定理モジュールを構築した. このシステムには一つのサーバと複数の利用者と物品が存在し, サーバは利用者に物品の貸出を行う. その結果をもとに, 定理モジュールの再利用可能性について検討する.

論文の構成

- 第二章：領域の分析モデルと理論

  この章では, まず, 本章で考える領域の概念を説明し, クラスモデルを定義する. そして, モデルが表現する領域に対し理論を導入する.

- 第三章：HOL 定理モジュールの構成

  前章で定義した領域の理論を HOL 上に実装する方法を示す.

- 第四章：例題

  貸出システムについて定理モジュールを構築手順を説明する.

- 第五章：考察

  例題の結果について分析を行う.

第2章　分析モデルと領域の理論

2.1　領域の捉え方

領域を協調動作するオブジェクトの集合と考える．オブジェクトは属性, リンク集合, 関数を持っている．リンクは, 他のオブジェクトとの間に張られるものであり, そのオブジェクトとなんらかの関係があることを示す．領域の状態は, 個々のオブジェクトの属性と, オブジェクト間に張られているリンクによって表現する．関数は, 属性の変更, リンクの張替え, オブジェクト生成を行うものであり, あるオブジェクトの関数呼出しを起点に, 複数のオブジェクトが関数を呼び出しあい, 領域の状態を変化させる．

図 2.1 は図書館領域を示したものである．上段は,ID100 の利用者が ID123 の本を借用しており,ID456 の本は誰にも借用されていない状態を表している. 利用者と本の間に張られるリンクによって借用中であることを表す．中段は, 上段の状態に対して「ID100 の利用者が ID456 の本を貸し出す」という関数が適用されたときの状態を表す.ID100 の利用者と ID456 の本にリンクが張られる．下段は, 中段の状態に対して「ID100 の利用者が ID123 の本の貸出を延長する」という関数が適用されたときの状態を表す.ID123 の本の返却日が 2 週間延長される．

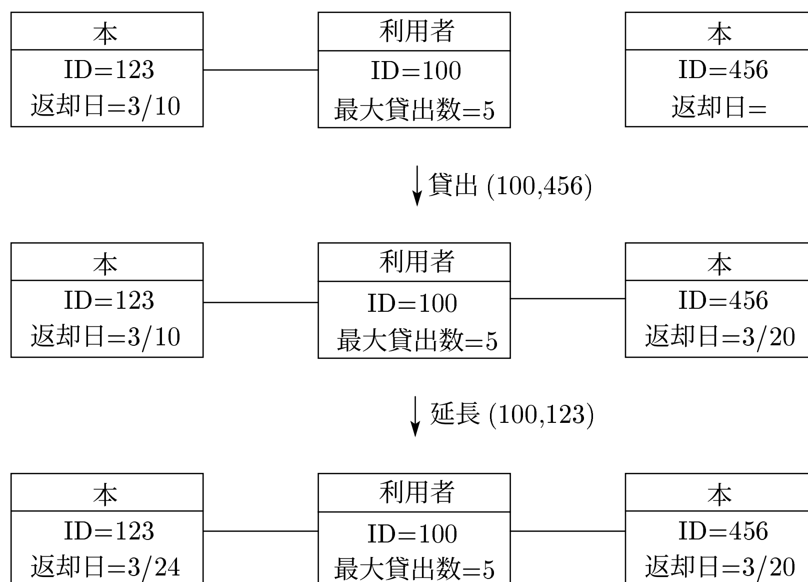

図 2.1: 領域の状態変化

2.2 クラスモデル

本研究では, 単純にするために, 分析モデルはクラスモデルのみを考える. その構成要素は, クラス, 属性, 操作, リンク集合とする. リンク集合は, その値がオブジェクトの集合である特別な属性とみなす.

定義 2.2.1 (クラスモデル) クラスモデルに導入する集合を以下に定める.

- $ClassID$：クラス識別子の集合
- $AttrID$：属性識別子の集合
- $LinksID$：リンク集合識別子の集合
- $FunID$：関数識別子の集合
- $Type$：型の集合

クラスモデルを以下に定義する.

$$CM = (C, A, L, F, \mathcal{M}, \mathcal{T})$$

ただし,

- $C \subseteq ClassID$
- $A \subseteq AttrID$
- $L \subseteq LinksID$
- $F \subseteq FunID$
- $\mathcal{M} = \mathcal{M}_{attr} \oplus \mathcal{M}_{links} \oplus \mathcal{M}_{fun}$
- $\mathcal{M}_{attr} : ClassID \to Pow(AttrID)$
- $\mathcal{M}_{links} : ClassID \to Pow(LinksID)$
- $\mathcal{M}_{fun} : ClassID \to Pow(FunID)$
- $\mathcal{T} = \mathcal{T}_{obj} \oplus \mathcal{T}_{attr} \oplus \mathcal{T}_{link} \oplus \mathcal{T}_{fun}$
- $\mathcal{T}_{obj} : ClassID \to Type$
- $\mathcal{T}_{attr} : AttrID \to Type$
- $\mathcal{T}_{link} : LinksID \to Type$
- $\mathcal{T}_{fun} : FunID \to Type$

とする. $\mathcal{M}_{attr}$ はクラス識別子とそれが持つ属性集合を対応付ける. $\mathcal{M}_{links}$ はクラス識別子とそれが持つリンク集合の集合を対応付ける. $\mathcal{M}_{fun}$ はクラス識別子とそれが持つ関数集合を対応付ける. $\mathcal{T}_{obj}$ はクラス識別子とそのクラスから実体化するオブジェクトの型を対応付ける. $\mathcal{T}_{attr}$ は属性識別子とその型を対応付ける. $\mathcal{T}_{links}$ はリンク集合識別子とそのリンク集合に属すリンク（オブジェクト）の型を対応付ける.$\mathcal{T}_{links}$ について以下の等式が成り立つ.

$$\mathcal{T}_{link}(s) = \mathcal{T}_{obj}(\mathcal{M}_{links}^{-1}(s))$$

$\mathcal{T}_{fun}$ は関数識別子とその関数の型を対応付ける.

2.3 領域の理論

クラスモデルから実体化する領域はオブジェクト集合から構成され, オブジェクトが互いの関数を呼び出しあうことにより領域の状態を変化させる. クラスモデルの定義にもとづき, この領域を表現する公理系を定める.

2.3.1 領域の型

領域は型 dom を持つとする. 直感的には領域内に存在するオブジェクトの集合を表す. これらのオブジェクトは上で定義したクラスから実体化するものである. 型 dom は定数として $init$ をもつ. これは領域の初期値, つまり, 初期に存在するオブジェクトの集合である.

2.3.2 領域のオペレータ

領域上のオペレータは以下の４つに分類される.

- 存在検査述語
- 基本関数
- オブジェクト生成関数
- 複合関数

存在検査述語

存在検査述語は領域内にオブジェクトが存在するかどうかを検査する述語である.

定義 2.3.1 (存在検査述語) 各クラス $c_i(i = 1..n)$ について以下の存在検査述語を導入する. ただし,$class_i$ は c_i の名前とする.

$$class_i_exists : \mathcal{T}_{obj}(c_i) \to dom \to bool$$

$class_i_exists\ o\ d$ はオブジェクト o が領域 d 内に存在することを意味する（実際の意味は後で導入する公理群によって与えられる）.

基本関数

基本関数はオブジェクトのデータを直接操作するオペレータである. 基本関数は, 属性操作関数, リンク集合操作関数の２つに分類される.

定義 2.3.2 (属性操作関数) クラス c が属性 $a_i(i = 1..n)$ を持つとき,c は以下の $2n$ 個の属性操作関数を持つ. ただし,$class$ は c の名前,$attr_i$ は a_i の名前とする.

$$class_get_attr_i : \mathcal{T}_{obj}(c) \to dom \to \mathcal{T}_{attr}(a_i)$$
$$class_set_attr_i : \mathcal{T}_{obj}(c) \to \mathcal{T}_{attr}(a_i) \to dom \to dom$$

各関数の意味は以下の通りである.

- $class_get_attr_i\ o\ d$

  オブジェクト o のもつ属性 a_i の値を取得する.

- $class_set_attr_i\ o\ v\ d$

  オブジェクト o のもつ属性 a_i の値を v に更新する.

定義 2.3.3 (リンク集合操作関数) クラス c がリンク集合 $s_i(i = 1..n)$ を持つとき, c は以下の $4n$ 個のリンク集合操作関数を持つ. ただし,$class$ は c の名前,$links_i$ は s_i の名前とする.

$$class_get_links_i_list : \mathcal{T}_{obj}(c) \to \mathcal{T}_{link}(s_i)\ list$$
$$class_init_links_i\ \mathcal{T}_{obj}(c) \to dom \to dom$$
$$class_add_links_i : \mathcal{T}_{obj}(c) \to \mathcal{T}_{link}(s_i) \to dom \to dom$$
$$class_del_links_i : \mathcal{T}_{obj}(c) \to \mathcal{T}_{link}(s_i) \to dom \to dom$$

各関数の意味は以下の通りである.

- $class_get_links_i_list\ o\ d$

  オブジェクト o のリンク集合 s_i の値をオブジェクトの配列として取得.

- $class_init_links_i\ o\ d$

  オブジェクト o のリンク集合 s_i の値を空集合とする.

- $class_add_links_i(p)\ o\ l\ d$

  オブジェクト o のリンク集合 s_i の値にリンク l を加える.

- $class_del_links_i(p)\ o\ l\ d$

  オブジェクト o のリンク集合 s_i の値からリンク l を削除する.

オブジェクト生成関数

オブジェクト生成関数は領域内に新たなオブジェクトを生成するオペレータである.

定義 2.3.4 (オブジェクト生成関数) クラスモデルに存在するすべてのクラス $c_i(i = 1..n)$ について以下のオブジェクト生成関数を定義する. ただし,$class_i$ は c_i の名前とする.

$$gen_class_i : dom \to \mathcal{T}_{obj}(c_i) * dom$$

gen_class_i は領域内に c_i のオブジェクトを生成し, そのオブジェクトを返す.

複合関数

定義 2.3.5 (複合関数) クラス c が関数 $f_i(i = 1..n)$ を持つとき, c について以下の n 個の複合関数を定義する. ただし,$class$ は c の名前,fun_i は f_i の名前とする.

$$class_fun_i$$

関数の型は,$\mathcal{T}_{fun}(f_i)$ が,

1. $'a \rightarrow 'b$

2. $unit \rightarrow 'b$

3. $'a \rightarrow unit$

4. $unit \rightarrow unit$

のとき, それぞれ,

1. $\mathcal{T}_{obj}(c) \rightarrow 'a \rightarrow dom \rightarrow 'b * dom$

2. $\mathcal{T}_{obj}(c) \rightarrow dom \rightarrow 'b * dom$

3. $\mathcal{T}_{obj}(c) \rightarrow 'a \rightarrow dom \rightarrow dom$

4. $\mathcal{T}_{obj}(c) \rightarrow dom \rightarrow dom$

とする. *1* について, 等式

$$f\ o\ in\ d = (out, d')$$

は,「関数 f が, オブジェクト o によって入力 in とともに起動され, 領域の状態を d から d' に変化させる. 同時に,out を出力する」を意味する. *2,3,4* は, それぞれ, 入力なし関数, 出力なし関数, 入出力なし関数である. *1,2* について, 関数が領域の状態を変化させないとき, つまり, 参照関数であるとき, 出力の dom は省略する. つまり, 以下の型とする.

- $\mathcal{T}_{obj}(c) \rightarrow 'a \rightarrow dom \rightarrow 'b$
- $\mathcal{T}_{obj}(c) \rightarrow dom \rightarrow 'b$

領域上のすべてのオペレータは上の 6 通りの型の形式を持つ. それぞれの形式をもつオペレータを $type1$,...,$type6$ のオペレータと呼ぶ.

複合関数の定義式

複合関数の定義式は以下を満たすように構成される.

クラス $c_i(i = 1..n)$ の基本関数の集合, 複合関数の集合をそれぞれ X_i,Y_i とし, 領域中のオブジェクト生成関数の集合を Z とすると, 複合関数 $class_i_fun_j$ の定義式は

$$X_i \cup \{\bigcup_i Y_i\} \cup Z$$

から構成される. また,$fst, cons, map$ などの標準的な関数も使用してよい.

このように複合関数の定義式を構成することにより, オブジェクトが関数を呼び出うという原理を公理系に実現することができる.

関数呼び出しに関して次の事実が存在する.

- 関数呼出しは一つの領域内で閉じる.
- あるクラスの関数が他のクラスの関数を呼び出すためには, そのクラスのオブジェクトを取得しなければならない. これは, すべての関数の第一引数はその関数を持つクラスのオブジェクトだからである. これには以下の４つの場合がある.
  - リンク集合から $class_get_links_list$ により得る.
  - オブジェクト生成関数を使う.
  - 関数の入力から得る.
  - 関数の中で呼び出した補助関数の出力から得る.

  この他に直接オブジェクトの定数を使うという方法があるが, 一般的な方法ではないので除外する.

2.3.3 公理

領域のオペレータについて, 以下の公理を導入する. 1 から 8 については,

$$class_i_exists\ o_1\ d \wedge class_j_exists\ o_2\ d$$

という前提条件が必要である.

[A-1] $class_get_attr$ と $class_set_attr$ の関係

$$\begin{aligned} & class_i_get_attr_k\ o_1\ (class_j_set_attr_l\ o_2\ v\ d) \\ = & \begin{cases} v, & i = j \text{ かつ } k = l \text{ かつ } o_1 = o_2 \text{ のとき} \\ class_i_get_attr_k\ o_1\ d, & \text{その他のとき} \end{cases} \end{aligned}$$

[A-2] $class_get_links_list$ と $class_init_links$ の関係

$$\begin{aligned} & class_i_get_links_k_list\ o_1\ (class_j_init_links_l\ o_2\ d) \\ = & \begin{cases} nil, & i = j \text{ かつ } k = l \text{ かつ } o_1 = o_2 \text{ のとき} \\ class_i_get_links_k_list\ o_1\ d, & \text{その他のとき} \end{cases} \end{aligned}$$

[A-3] $class_get_links_list$ と $class_add_links$ の関係

$$\begin{aligned} & class_i_get_links_k_list\ o_1\ (class_j_add_links_l\ o_2\ a\ d) \\ = & \begin{cases} add\ a\ (class_i_get_links_k_list\ o_1\ d), & i = j \text{ かつ } k = l \text{ かつ } o_1 = o_2 \text{ のとき} \\ class_i_get_links_k_list\ o_1\ d, & \text{その他のとき} \end{cases} \end{aligned}$$

[A-4] $class_get_links_list$ と $class_del_links$ の関係

$$class_i_get_links_k_list\ o_1\ (class_j_del_links_l\ o_2\ a\ d) = \begin{cases} del\ a\ (class_i_get_links_k_list\ o_1\ d), & i = j \text{ かつ } k = l \text{ かつ } o_1 = o_2 \text{ のとき} \\ class_i_get_links_k_list\ o_1\ d, & \text{その他のとき} \end{cases}$$

[A-5] $class_get_attr$ と $class_init_link$ の関係

$$class_i_get_attr_k\ o_1\ (class_j_init_link_l\ o_2\ d) = class_i_get_attr_k\ o_1\ d$$

[A-6] $class_get_attr$ と $class_add_link$ の関係

$$class_i_get_attr_k\ o_1\ (class_j_add_link_l\ o_2\ a\ d) = class_i_get_attr_k\ o_1\ d$$

[A-7] $class_get_attr$ と $class_del_link$ の関係

$$class_i_get_attr_k\ o_1\ (class_j_del_link_l\ o_2\ a\ d) = class_i_get_attr_k\ o_1\ d$$

[A-8] $class_get_links_list$ と $class_set_attr$ の関係

$$class_i_get_links_k_list\ o_1\ (class_j_set_attr_l\ o_2\ a\ d) = class_i_get_links_k_list\ o_1\ d$$

[A-9] $class_get_attr$ と gen_class の関係

$$(o', d') = gen_class_i\ d \text{ とすると,}$$
$$class_i_exists\ o\ d \Rightarrow (class_i_get_attr_k\ o\ d' = class_i_get_attr_k\ o\ d)$$

[A-10] $class_get_links_list$ と gen_class の関係

$$(o', d') = gen_class_i\ d \text{ とすると,}$$
$$class_i_exists\ o\ d \Rightarrow (class_i_get_links_k_list\ o\ d' = class_i_get_links_k_list\ o\ d)$$

[A-11] $class_exists$ と gen_class の関係（1）

$$(o, d') = gen_class_i\ d \text{ とすると,}$$
$$class_i_exists\ o\ d' \wedge \neg(class_i_exists\ o\ d)$$

[A-12] $class_exists$ と gen_class の関係（2）

$$(o', d') = gen_class_i\ d \text{ とすると,}$$
$$class_i_exists\ o\ d \Rightarrow class_i_exists\ o\ d'$$

[A-13] $class_exists$ と $class_set_attr$ の関係

$$class_i_exists\ o_1\ d = class_i_exists\ o_1\ (class_j_set_attr_k\ o_2\ v\ d)$$

[A-14] $class_exists$ と $class_init_links$ の関係

$$class_i_exists\ o_1\ d = class_i_exists\ o_1\ (class_j_init_links_k\ o_2\ a\ d)$$

[A-15] $class_exists$ と $class_add_links$ の関係

$$class_i_exists\ o_1\ d = class_i_exists\ o_1\ (class_j_add_links_k\ o_2\ a\ d)$$

[A-16] $class_exists$ と $class_del_links$ の関係

$$class_i_exists\ o_1\ d = class_i_exists\ o_1\ (class_j_del_links_k\ o_2\ a\ d)$$

[A-17] $class_get_link_list$ の性質

$$l = class_i_get_links_j_list\ o\ d\ \text{とすると,}$$
$$is_el\ x\ l \Rightarrow (count_el\ x\ l = 1)$$

[A-18] $class_fun$ の定義

$$class_i_fun_j = definition_{ij}$$

各公理の意味を以下に示す.

[A-1] 左辺は, オブジェクト o_2 の属性 a_l を v に更新した直後のオブジェクト o_1 から取得される属性 a_k の値を表す. この二つのオブジェクトが同一のオブジェクトであり（つまり,$o_1 = o_2$ のとき. このとき $i = j$ も成り立つ）, 同一の属性を更新, 取得するとき（つまり,$k = l$ のとき）, 取得される属性の値は v である. それ以外のときは, 更新前に取得する属性の値と等しい. つまり,o_2 の持つ属性 a_l 以外の属性は影響を受けない.

[A-2] 左辺は, オブジェクト o_2 のリンク集合 s_l を空集合に初期化した直後のオブジェクト o_1 から取得されるリンク集合 s_k の値を表す.　同一のオブジェクトが, 同一のリンク集合を初期化, 取得するとき, 取得されるリンク集合の値は nil である. それ以外のときは, 初期化前に取得するリンク集合の値と等しい. つまり,o_2 の持つリンク集合 a_l 以外のリンク集合は影響を受けない.

[A-3] 同一のオブジェクトが, 同一のリンク集合に対して追加, 取得するとき, 取得されるリンク集合の値は, 追加前に取得するリンク集合に $add\ a$ を適用した値である.o_2 の持つリンク集合 a_l 以外のリンク集合は影響を受けない. ただし,$add : 'a \rightarrow 'a\ list \rightarrow 'a\ list$ は以下に定義される関数である.

$$add\ a\ l = \begin{cases} cons\ a\ l, & \neg(is_el\ a\ l)\ \text{のとき} \\ l, & is_el\ a\ l\ \text{（}a\ \text{が}\ l\ \text{の要素）のとき} \end{cases}$$

[A-4] 同一のオブジェクトが, 同一のリンク集合に対して削除, 取得するとき, 取得される属性の値は, 削除前に取得するリンク集合に $del\ a$ を適用した値である.o_2 の持つリンク集合 a_l 以外のリンク集合は影響を受けない. ただし,$del : 'a \rightarrow 'a\ list \rightarrow 'a\ list$ は以下に定義される関数である.

$$\begin{aligned} del\ a\ nil &= nil \\ del\ a\ (cons\ x\ l) &= \begin{cases} l, & a = x\ \text{のとき} \\ del\ a\ l, & \neg(a = x)\ \text{のとき} \end{cases} \end{aligned}$$

[A-5] オブジェクト o_2 のリンク集合 s_l を初期化した直後, オブジェクト o_1 の属性 s_k を取得すると, その値は更新前に取得する値と等しい. つまり, リンク集合の初期化は, 属性に影響を与えない.

[A-6] リンク集合に対する要素の追加は, 属性に影響を与えない.

[A-7] リンク集合に対する要素の削除は, 属性に影響を与えない.

[A-8] オブジェクト o_2 の属性を a に更新した直後, オブジェクト o_1 のリンク集合 s_k を取得すると, その値は更新前に取得する値と等しい. つまり, 属性値の更新は, リンク集合に影響を与えない.

[A-9] オブジェクト o が領域 d 内に存在するならば, クラス c_j のオブジェクトが生成された直後に o から取得される属性 a_k の値は, 生成前に取得する値と等しい. つまり, 新たなオブジェクトが生成されてもすでに存在しているオブジェクトの属性は影響を受けない.

[A-10] オブジェクト o が領域 d に存在するならば, クラス c_j のオブジェクトが生成された直後に o から取得されるリンク集合 l_k の値は, 生成前に取得する値と等しい. つまり, 新たなオブジェクトが生成されてもすでに存在しているオブジェクトのリンク集合は影響を受けない.

[A-11] クラス c_i から生成されたオブジェクト o は, 生成直後の領域 d' に存在する. また, 生成前の領域 d には存在しない.

[A-12] オブジェクト o が領域に存在するならば, あるクラスのオブジェクトが生成された直後も領域に存在する.

[A-13] オブジェクト o が領域に存在するならば, あるオブジェクトの属性値が更新された直後も領域に存在する.

[A-14] オブジェクト o が領域に存在するならば, あるオブジェクトのリンク集合が初期化された直後も領域に存在する.

[A-15] オブジェクト o が領域に存在するならば, あるオブジェクトのリンク集合にリンクが追加された直後も領域に存在する.

[A-16] オブジェクト o が領域に存在するならば, あるオブジェクトのリンク集合からリンクが削除された直後も領域に存在する.

[A-17] リンク集合にはリンクが重複して含まれない.$count_el\ x\ l$ はリスト l に含まれる要素 x の個数を返す関数である.

[A-18] 関数の定義.$definition_{ij}$ はその定義式を表す.

第3章　定理モジュールの構成

3.1　定理モジュール

ＨＯＬにおける理論は次の４つの要素から構成される.

- 型
- 定数
- 公理
- 定理

ＨＯＬに存在する最も原始的な理論は PPLAMB と呼ばれ, ＬＣＦの理論が基礎となっている. ＨＯＬに存在するすべての理論はこの PPLAMB を起点とする拡張の繰り返しにより生成されている（図 3.1）.

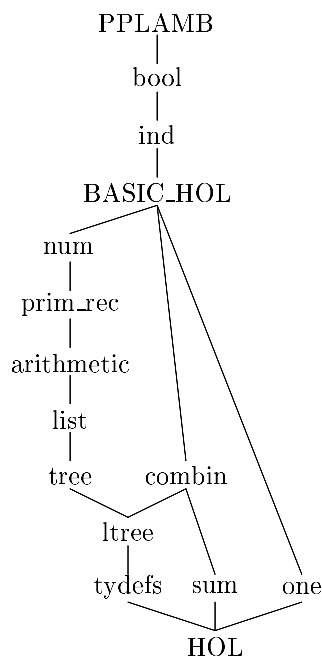

図 3.1: 理論の構成

ユーザに初期に与えられる理論は HOL というものであり, ブール代数, 自然数, リストの理論といった基本的な理論を含んでいる. この初期理論を拡張していくことにより, 新たな理論を生成していくことが可能である. 拡張により生成される複数の理論は階層的に構成され, 親子関係をもつ.

一つの理論は, 一つのファイルによって代表される. このファイルは理論ファイルと呼ばれ, 上に挙げた４つの要素の他に, 親の理論ファイルへのポインタ情報や, 定数に関する prefix,infix ステータスなどの付加情

報を含んでいる. 一つの理論ファイルによって表される理論は, そのファイルと, そのファイルからポインタによって到達可能なすべてのファイルに含まれる型, 定数, 公理, 定理を集めたものである. 本研究では, 個々の理論ファイルによって代表される理論を, 定理モジュールと呼ぶ.

3.2 定理モジュール構成の概要

領域の公理系をＨＯＬでは既存の理論からの事実の積み上げにより構成する. 前章で定めた公理系においては, 複数のクラスから構成される領域を一つの公理系で表現しており, すべてのクラスのすべての基本関数について, その関係を定めている. その公理の数はかなりのものである. しかし, 実際に定理を証明する際に必要となる公理はその一部である. したがって, 実装において, 複数のクラスから構成される領域を一つの定理モジュールから構成するのは, 不必要な公理を生成することにつながり, 有効ではない. また, 一つの定理モジュールで構成してしまうと, その理論の中で再利用可能な要素を抽出するのが難しい.

関数は関数呼出しが領域内で閉じるように定義される. 逆に考えれば, 関数呼出しが閉じるクラス群は一つの領域と考えることができる. 一つの領域を構築する際, 最初は定理モジュールをクラスごとに生成していく. 複数のクラスに及ぶ関数呼出しを定義する際は, それらのクラスの定理モジュールを統合して一つの定理モジュールを作り, その中で関数を定義する.

実装においては, まず, 一般的なクラスのオペレータを含む定理モジュールを構築した. これは,obj1 から

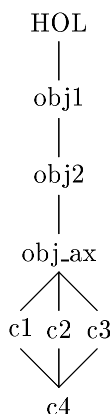

図 3.2: 定理モジュールの構成

obj_ax にかけて, 段階的に構成していく. obj_ax に含まれる基本関数は, 定義を一般的なものとするために高階関数になっている. また, この基本関数は, 前章で定義したような領域全体を入出力に取るものではなく, 一つのクラスに属すオブジェクト集合のみを入出力に取るものである. obj_ax 以降は, 構成の仕方が領域に依存する.

各定理モジュールの概略を以下に示す.

- obj1：一つのクラスのみを考え, そのオブジェクトとオブジェクト集合を表現する. ここでは, オブジェクト集合に対して, オブジェクトの取得, 更新, 生成を行う関数を定義する. また, それらに関する定理を証明する.

- obj2：obj1 で定義された関数を用いて, 基本関数の一般形を定義する. また, それらに関する定理を証明する.

- obj_ax：obj2 で定義された基本関数の一般形から, 属性操作関数, リンク集合操作関数の一般形を定義する. また, それらに関する定理を証明する.

本章では,obj1,obj2,obj_ax の内容を説明する．それ以降は, 領域に依存するので, 次章の例題の中で説明する.

3.3 obj1

3.3.1 定義

この定理モジュールでは, オブジェクトとオブジェクト集合の基本的な概念を構築する.

オブジェクト集合は, オブジェクトのデータのメモリであると考える. オブジェクトは, メモリ上のポインタである. オブジェクトはそのポインタによってメモリ上の自分のデータを識別する．オブジェクトのデータは属性, リンク集合からなる. このデータの型を'a とする．オブジェクト集合は, このデータの配列と考える. つまり, 型は'a list である．オブジェクト識別子（ポインタ）は, その配列上の位置とする.

オブジェクト集合上の 3 個の関数と 1 個の述語を以下に定義する.

- GET_OBJ

  オブジェクト集合 l から指定された識別子を持つオブジェクトを取得する.

```
|- (!(l:'a list). (GET_OBJ 0 l = HD l)) /\
   (!n l. GET_OBJ (SUC n) l = GET_OBJ n (TL l))
```

- SET_OBJ

  オブジェクト集合 l 中の指定された識別子を持つオブジェクトの値を x に設定する. 存在しなければ何も行わない.

```
|- (!(x:'a) l. (SET_OBJ1 0 x l = CONS x (TL l))) /\
   (!n (x:'a) l.
    SET_OBJ1 (SUC n) x l = CONS (HD l) (SET_OBJ1 n x (TL l)))

|- !i (x:'a) l. SET_OBJ i x l = (EXISTS i l => SET_OBJ1 i x l | l)
```

- GEN_OBJ

  オブジェクト集合 l に, 値 x を持つ新たなオブジェクトを生成し, オブジェクトの識別子を出力する.SNOC は CONS と対称の動作を行う関数であり, リストの末尾に要素を追加する.

```
|- !(x:'a) l. GEN_OBJ x l = let i = LENGTH l in (i,SNOC x l)
```

- EXISTS

  オブジェクト i がオブジェクト集合 l に存在するということは, 識別子の値がリスト長より小さいということである.

```
|- !i (l:'a list). EXISTS i l = i < LENGTH l
```

3.3.2 定理

- GET_OBJ と SET_OBJ の関係

```
GET_OBJ_SET_OBJ1
|- (x:'a) i l. EXISTS i l ==> (GET_OBJ i (SET_OBJ i x l) = x)

GET_OBJ_SET_OBJ2
|- !(x:'a) i j l.
      ~(i = j) ==> (GET_OBJ i (SET_OBJ j x l) = GET_OBJ i l)
```

  - GET_OBJ_SET_OBJ1
    オブジェクト i の値を x に設定した後, 同じオブジェクトを取得すれば, その値は x である. ただし,i がオブジェクト集合 l 中に存在していることが条件である.
  - GET_OBJ_SET_OBJ2
    オブジェクト i の値を x に更新した後,i とは異なるオブジェクト j を取得すれば, その値は更新前に取得する値と等しい.

- EXISTS と SET_OBJ の関係

```
EXISTS_SET_OBJ
|- !(x:'a) i j l. EXISTS i l = EXISTS i (SET_OBJ j x l)
```

オブジェクト i がオブジェクト集合 l 中に存在するならば, オブジェクト j の値を更新した後も存在する.

- GET_OBJ と GEN_OBJ の関係

```
GET_OBJ_GEN_OBJ_FST_SND
|- !(x:'a) l. let p = GEN_OBJ x l in GET_OBJ (FST p) (SND p) = x

GET_OBJ_GEN_OBJ
|- !(x:'a) i l. let p = GEN_OBJ x l in
   EXISTS i l ==> (GET_OBJ i (SND p) = GET_OBJ i l)
```

  - GET_OBJ_GEN_OBJ_FST_SND
    値 x を持つオブジェクトを生成した直後, そのオブジェクトを取得すれば, その値は x である.
  - GET_OBJ_GEN_OBJ

オブジェクト i がオブジェクト集合 l 中に存在するならば, 新たなオブジェクトが生成された直後に取得する i の値は, 生成前に取得する値と等しい.

- EXISTS と GEN_OBJ の関係.

```
EXISTS_GEN_OBJ_FST_SND
|- !(x:'a) l. let p = GEN_OBJ x l in
   EXISTS (FST p) (SND p) /\ ~(EXISTS (FST p) l)

EXISTS_GEN_OBJ
|- !(x:'a) l. let p = GEN_OBJ x l in
   EXISTS (FST p) (SND p) /\ ~(EXISTS (FST p) l)

EXISTS_IMP_NOT_FST_GEN_OBJ
|- !i (x:'a) l. EXISTS i l ==> ~(i = FST (GEN_OBJ x l))
```

  - EXISTS_GEN_OBJ_FST_SND

    新たに生成したオブジェクトはオブジェクト集合中に存在する. また, 生成前のオブジェクト集合中には存在しない.

  - EXISTS_GEN_OBJ

    オブジェクト i がオブジェクト集合 l 中に存在するならば,GEN_OBJ によって新たなオブジェクトを生成した後も存在する.

  - EXISTS_IMP_NOT_FST_GEN_OBJ

    新たに生成されたオブジェクトの識別子は, 生成前に存在していたすべてのオブジェクトの識別子と異なる. この定理は上の二つの定理から導くことができる.

3.4 obj2

3.4.1 定義

この定理モジュールは obj1 の関数を用いて, オブジェクトのデータに含まれる要素（属性, リンク集合）を操作する関数を定義する. オブジェクトのデータはその要素が組になったものである（型はまだ'a のままとしておく）. 任意の要素を操作する関数とするために以下で定義する関数は高階関数としてある.

- OBJ_GET

  オブジェクト i の n 番目の要素を取得する. 何番目の要素を取得するかは, 関数変数 f に代入される関数によって決まる.

```
|- !i (ol:'a list) (f:'a->'b). OBJ_GET f i ol = f (GET_OBJ i ol)
```

- OBJ_SET

オブジェクト i の n 番目の要素を a に更新する. 何番目の要素を更新するかは, 関数変数 g に代入される関数によって決まる.

```
|- !i (ol:'a list) (a:'b) (g:'b->'a->'a).
   OBJ_SET g i a ol = SET_OBJ i (g a (GET_OBJ i ol)) ol
```

例えば, データの要素が３つであるオブジェクトに対して各要素を操作する関数は次のように定義される。

- １番目の要素を取得, 更新

```
OBJ_GET1 i ol   = OBJ_GET GET1 i ol
OBJ_SET1 i x ol = OBJ_GET SET1 i x ol
```

- ２番目の要素を取得, 更新

```
OBJ_GET2 i ol   = OBJ_GET GET2 i ol
OBJ_SET2 i x ol = OBJ_GET SET2 i x ol
```

- ３番目の要素を取得, 更新

```
OBJ_GET3 i ol   = OBJ_GET GET3 i ol
OBJ_SET3 i x ol = OBJ_GET SET3 i x ol
```

ただし,

```
GET1 (a,b,c) = a
GET2 (a,b,c) = b
GET3 (a,b,c) = c
SET1 x (a,b,c) = (x,b,c)
SET2 x (a,b,c) = (a,x,c)
SET3 x (a,b,c) = (a,b,x)
```

である.

OBJ_GET,OBJ_SET の二つの関数変数 f,g について２つの条件を定義する.

```
COND1
|- !(f:'a->'b) (g:'b->'a->'a). COND1 f g = (!s t. f (g s t) = s)

COND2
|- !(f:'a->'b) (g:'c->'a->'a). COND2 f g = (!s t. f (g s t) = f t)
```

COND1 f g が成り立つとき,OBJ_GET f と OBJ_SET g は同じ要素に対する取得, 更新を意味する. また,COND2 f g が成り立つときは, 異なる要素に対する取得, 更新を意味する. 上の例では,

```
|- COND1 GET1 SET1 /\ COND1 GET2 SET2 /\ COND1 GET3 SET3 /\
   COND2 GET1 SET2 /\ COND2 GET1 SET3 /\
   COND2 GET2 SET1 /\ COND2 GET2 SET3 /\
   COND2 GET3 SET1 /\ COND2 GET3 SET2
```

である.

3.4.2 定理

- OBJ_GET と OBJ_SET の関係

```
OBJ_GET_OBJ_SET1
|- !i (ol:'a list) (f:'a->'b) g a.
   EXISTS i ol ==> COND1 f g ==> (OBJ_GET f i (OBJ_SET g i a ol) = a)

OBJ_GET_OBJ_SET2
|- !i (ol:'a list) (f:'a->'b) g (a:'c). EXISTS i ol ==> COND2 f g ==>
   (OBJ_GET f i (OBJ_SET g i a ol) = OBJ_GET f i ol)

OBJ_GET_OBJ_SET3
|- !i j (ol:'a list) (f:'a->'b) g (a:'c).
   ~(i = j) ==> (OBJ_GET f i (OBJ_SET g j a ol) = OBJ_GET f i ol)
```

  – OBJ_GET_OBJ_SET1
    オブジェクト i がある要素を a に更新した直後,i が同じ要素を取得するとき, その値は a である. ただし,i がオブジェクト集合 ol 中に存在することが条件である.
  – OBJ_GET_OBJ_SET2
    オブジェクト i がある要素を更新した直後,i が別の要素を取得するとき, その値は更新前に取得する値と等しい. ただし,i がオブジェクト集合 ol 中に存在することが条件である.
  – OBJ_GET_OBJ_SET3
    オブジェクト j がある要素を更新した直後,j とは異なるオブジェクト i が取得する要素の値は, 更新前に取得する値と等しい.

- EXISTS と OBJ_SET の関係

```
EXISTS_OBJ_SET
|- !i j g (x:'b) (ol:'a list).
   EXISTS i ol = EXISTS i (OBJ_SET g j x ol)
```

オブジェクト i がオブジェクト集合 ol 中に存在するならば, オブジェクト j がある要素を更新した後も i は存在する.

- GET_OBJ と GEN_OBJ の関係

```
OBJ_GET_GEN_OBJ_FST_SND
|- !x (ol:'a list) (f:'a->'b).
   let p = GEN_OBJ x ol in OBJ_GET f (FST p) (SND p) = f x

OBJ_GET_GEN_OBJ
|- !i x (ol:'a list) (f:'a->'b). let p = GEN_OBJ x ol in
   EXISTS i ol ==> (OBJ_GET f i (SND p) = OBJ_GET f i ol)
```

– OBJ_GET_GEN_OBJ

  値 x を持つ新たなオブジェクトを生成した直後, そのオブジェクトの n 番目の要素を取得すると, その値は x の n 番目の要素である.

– OBJ_GET_GEN_OBJ

  オブジェクト i がオブジェクト集合 ol 中に存在するならば, 新たなオブジェクトが生成された直後,i の要素を取得すると, その値は生成前に取得する値と等しい.

3.5 obj_ax

3.5.1 定義

この定理モジュールでは,obj2 の関数を用いて基本関数の一般形を定義する. また, それらの間に成り立つ定理を証明する.

obj2 ではオブジェクトのデータの型は'a であったが, obj_ax ではそれを'a#'b の組とする。'a によりリンク集合（オブジェクト識別子のリスト）の組,'b により属性の組を表現する. 以下, それぞれを, リンク部, 属性部と呼ぶ.

オブジェクト集合は ('a#'b)list の型をもつ. この集合に対し, この集合に属すオブジェクトのリンク部だけを集めた集合（型は'a list）を得る関数, 属性部だけを集めた集合（型は'b list）を得る関数, および, それらを更新する関数を以下に定義する.

```
GET_LINK
|- !(ol:('a#'b)list). GET_LINK ol = UNZIP_FST ol

GET_ATTR
|- !(ol:('a#'b)list). GET_ATTR ol = UNZIP_SND ol

SET_LINK
|- !(l:'a list) (ol:('a#'b)list). SET_LINK l ol = ZIP (l,UNZIP_SND ol)

SET_ATTR
|- !(l:'b list) (ol:('a#'b)list). SET_ATTR l ol = ZIP (UNZIP_FST ol,l)
```

- GET_LINK：リンク部を得る

- GET_ATTR：属性部を得る
- SET_LINK：リンク部を更新
- SET_ATTR：属性部を更新

オブジェクト集合に対し,GET_LINK,GET_ATTR を適用することによって得られたリンク部, 属性部の集合に対し,obj2 の関数を適用する. 適用後,SET_LINK,SET_ATTR により更新後のオブジェクト集合を得る.

属性操作関数の定義

上で定義した関数を補助関数とし, 属性操作関数を定義する.

```
OBJ_GET_ATTR
|- !(f:'b->'c) i (ol:('a#'b)list).
   OBJ_GET_ATTR f i ol = OBJ_GET f i (GET_ATTR ol)

OBJ_SET_ATTR
|- !g i (a:'c) (ol:('a#'b)list).
   OBJ_SET_ATTR g i a ol =
   let l = OBJ_SET g i a (GET_ATTR ol) in SET_ATTR l ol
```

- OBJ_GET_ATTR

  オブジェクト i の n 番目の属性を取得.

- OBJ_SET_ATTR

  オブジェクト i の n 番目の属性を a に更新.

上の 2 つの関数から次の関数が定義できる.

```
OBJ_FUN_ATTR
|- !(f:'b->'c) g h i (ol:('a#'b)list).
   OBJ_FUN_ATTR f g h i ol =
   (COND1 f g => OBJ_SET_ATTR g i (h (OBJ_GET_ATTR f i ol)) ol | ol)
```

これはオブジェクト i の n 番目の属性に関数 h を適用する関数である. OBJ_GET_LINK によってリンク集合を取得し, その値に h を適用したものを OBJ_SET_LINK によって戻すという定義になっている. この取得と更新は同じリンク集合に対するものでなければ意味がないので,COND1 f g が条件である.

リンク集合操作関数の定義

まず, 以下の 3 つの関数を定義する. リンク集合の型は num list である. 一つのリンク集合に一つのオブジェクト識別子が重複して含まれないように, 関数 TO_SET を適用している.TO_SET はリスト中の重複する要素を 1 つにする関数である.

```
OBJ_GET_LINK
|- !(f:'a->num list) i (ol:('a#'b)list).
   OBJ_GET_LINK f i ol = TO_SET (OBJ_GET f i (GET_LINK ol))

OBJ_SET_LINK
|- !g i (a:num list) (ol:('a#'b)list).
   OBJ_SET_LINK g i a ol =
   let l = OBJ_SET g i (TO_SET a) (GET_LINK ol) in SET_LINK l ol

OBJ_FUN_LINK
|- !(f:'a->num list) g h i (ol:('a#'b)list).
   OBJ_FUN_LINK f g h i ol =
   (COND1 f g => OBJ_SET_LINK g i (h (OBJ_GET_LINK f i ol)) ol | ol)
```

- `OBJ_GET_LINK`

  オブジェクト i の n 番目のリンク集合を取得する.

- `OBJ_SET_LINK`

  オブジェクト i の n 番目のリンク集合を `TO_SET a` に更新する.

- `OBJ_FUN_LINK`

  オブジェクト i の n 番目のリンク集合に関数 h を適用する.

リンク集合に対して行える操作は, 初期化, リンクの追加, リンクの削除の 3 つに限定する. それぞれを行う関数を, 上で定義した `OBJ_SET_LINK`,`OBJ_FUN_LINK` を用いて定義する.

```
OBJ_INIT_LINK
|- !g i (ol:('a#'b)list). OBJ_INIT_LINK g i ol = OBJ_SET_LINK g i [] ol

OBJ_ADD_LINK
|- !(f:'a->num list) g i j (ol:('a#'b)list). OBJ_ADD_LINK f g i j ol =
   OBJ_FUN_LINK f g (CONS j) i ol

OBJ_DEL_LINK
|- !(f:'a->num list) g i j (ol:('a#'b)list). OBJ_DEL_LINK f g i j ol =
   OBJ_FUN_LINK f g (DEL j) i ol
```

- `OBJ_INIT_LINK`

  オブジェクト i の n 番目のリンク集合を初期化する.

- `OBJ_ADD_LINK`

  オブジェクト i の n 番目のリンク集合に新たなリンク j を追加する.

- OBJ_DEL_LINK

  オブジェクト i の n 番目のリンク集合からリンク j を削除する.

以上で定義した,

OBJ_GET_LINK
OBJ_INIT_LINK
OBJ_ADD_LINK
OBJ_DEL_LINK

がリンク集合操作関数の一般形である.

3.5.2 定理

以下に領域のオペレータをまとめる.

- 存在検査述語：EXISTS
- オブジェクト生成関数：GEN_OBJ
- 基本関数
  - 属性操作関数

    GET_OBJ SET_OBJ FUN_OBJ
  - リンク集合操作関数

    OBJ_GET_LINK OBJ_INIT_LINK OBJ_ADD_LINK OBJ_DEL_LINK

これらについて前章で定めた公理系を証明する.

リンク集合操作関数に関する定理

[B-1] OBJ_GET_ATTR と OBJ_SET_ATTR の関係

```
OBJ_GET_ATTR_OBJ_SET_ATTR1
|- !f g i (a:'c) (ol:('a#'b)list). EXISTS i ol ==> COND1 f g ==>
   (OBJ_GET_ATTR f i (OBJ_SET_ATTR g i a ol) = a)

OBJ_GET_ATTR_OBJ_SET_ATTR2
|- !(f:'b->'c) g i (a:'d) (ol:('a#'b)list).
   EXISTS i ol ==> COND2 f g ==>
   (OBJ_GET_ATTR f i (OBJ_SET_ATTR g i a ol) = OBJ_GET_ATTR f i ol)

OBJ_GET_ATTR_OBJ_SET_ATTR3
|- !(f:'b->'c) g i j (a:'d) (ol:('a#'b)list). ~(i = j) ==>
   (OBJ_GET_ATTR f i (OBJ_SET_ATTR g j a ol) = OBJ_GET_ATTR f i ol)
```

- OBJ_GET_ATTR_OBJ_SET_ATTR1

  A-1 の $i = j$ かつ $k = l$ かつ $o_1 = o_2$ のとき
- OBJ_GET_ATTR_OBJ_SET_ATTR2

  A-1 の $i = j$ かつ $\neg(k = l)$ かつ $o1 = o2$ のとき
- OBJ_GET_ATTR_OBJ_SET_ATTR3

  A-1 の $i = j$ かつ $\neg(o1 = o2)$ のとき

[B-2] OBJ_GET_LINK と OBJ_INIT_LINK の関係

```
OBJ_GET_LINK_OBJ_INIT_LINK1
|- !f g i (ol:('a#'b)list). EXISTS i ol ==> COND1 f g ==>
   (OBJ_GET_LINK f i (OBJ_INIT_LINK g i ol) = [])

OBJ_GET_LINK_OBJ_INIT_LINK2
|- !f g i (ol:('a#'b)list). EXISTS i ol ==> COND2 f g ==>
   (OBJ_GET_LINK f i (OBJ_INIT_LINK g i ol) = OBJ_GET_LINK f i ol)

OBJ_GET_LINK_OBJ_INIT_LINK3
|- !f g i j (ol:('a#'b)list). ~(i = j) ==>
   (OBJ_GET_LINK f i (OBJ_INIT_LINK g j ol) = OBJ_GET_LINK f i ol)
```

- OBJ_GET_LINK_OBJ_INIT_LINK1

  A-2 の $i = j$ かつ $k = l$ かつ $o_1 = o_2$ のとき
- OBJ_GET_LINK_OBJ_INIT_LINK2

  A-2 の $i = j$ かつ $\neg(k = l)$ かつ $o1 = o2$ のとき
- OBJ_GET_LINK_OBJ_INIT_LINK2

  A-2 の $i = j$ かつ $\neg(o1 = o2)$ のとき

[B-3] OBJ_GET_LINK と OBJ_ADD_LINK の関係

```
OBJ_GET_LINK_OBJ_ADD_LINK1
|- !f g i j (ol:('a#'b)list). EXISTS i ol ==> COND1 f g ==>
   (OBJ_GET_LINK f i (OBJ_ADD_LINK f g i j ol) =
   let l = OBJ_GET_LINK f i ol in (IS_EL j l => l | CONS j l))

OBJ_GET_LINK_OBJ_ADD_LINK2
|- !f1 f2 g i j (ol:('a#'b)list). EXISTS i ol ==> COND2 f1 g ==>
   (OBJ_GET_LINK f1 i (OBJ_ADD_LINK f2 g i j ol) =
    OBJ_GET_LINK f1 i ol)

OBJ_GET_LINK_OBJ_ADD_LINK3
|- !f1 f2 g i j k (ol:('a#'b)list). ~(i = j) ==>
   (OBJ_GET_LINK f1 i (OBJ_ADD_LINK f2 g j k ol) =
    OBJ_GET_LINK f1 i ol)
```

- OBJ_GET_LINK_OBJ_ADD_LINK1
  A-3 の $i = j$ かつ $k = l$ かつ $o_1 = o_2$ のとき
- OBJ_GET_LINK_OBJ_ADD_LINK2
  A-3 の $i = j$ かつ $\neg(k = l)$ かつ $o1 = o2$ のとき
- OBJ_GET_LINK_OBJ_ADD_LINK3
  A-3 の $i = j$ かつ $\neg(o1 = o2)$ のとき

[B-4] OBJ_GET_LINK と OBJ_DEL_LINK の関係

```
OBJ_GET_LINK_OBJ_DEL_LINK1
|- !f g i j (ol:('a#'b)list). EXISTS i ol ==> COND1 f g ==>
   (OBJ_GET_LINK f i (OBJ_DEL_LINK f g i j ol) =
   let l = OBJ_GET_LINK f i ol in (IS_EL j l => DEL j l | l))

OBJ_GET_LINL_OBJ_DEL_LINK2
|- !f1 f2 g i j (ol:('a#'b)list). EXISTS i ol ==> COND2 f1 g ==>
   (OBJ_GET_LINK f1 i (OBJ_DEL_LINK f2 g i j ol) =
    OBJ_GET_LINK f1 i ol)

OBJ_GET_LINK_OBJ_DEL_LINK3
|- !f1 f2 g i j k (ol:('a#'b)list). ~(i = j) ==>
   (OBJ_GET_LINK f1 i (OBJ_DEL_LINK f2 g j k ol) =
    OBJ_GET_LINK f1 i ol)
```

- OBJ_GET_LINK_OBJ_DEL_LINK1
  A-4 の $i = j$ かつ $k = l$ かつ $o_1 = o_2$ のとき

- `OBJ_GET_LINK_OBJ_DEL_LINK2`

  A-4 の $i = j$ かつ $\neg(k = l)$ かつ $o1 = o2$ のとき

- `OBJ_GET_LINK_OBJ_DEL_LINK3`

  A-4 の $i = j$ かつ $\neg(o1 = o2)$ のとき

[B-5] OBJ_GET_ATTR と OBJ_INIT_LINK の関係

```
OBJ_GET_ATTR_OBJ_INIT_LINK
|- !(f:'b->'c) g i j (ol:('a#'b)list).
   OBJ_GET_ATTR f i (OBJ_INIT_LINK g j ol) = OBJ_GET_ATTR f i ol
```

[B-6] OBJ_GET_ATTR と OBJ_ADD_LINK の関係

```
OBJ_GET_ATTR_OBJ_ADD_LINK
|- !(f1:'b->'c) f2 g i j k (ol:('a#'b)list).
   OBJ_GET_ATTR f1 i (OBJ_ADD_LINK f2 g j k ol) = OBJ_GET_ATTR f1 i ol
```

[B-7] OBJ_GET_ATTR と OBJ_DEL_LINK の関係

```
OBJ_GET_ATTR_OBJ_DEL_LINK
|- !(f1:'b->'c) f2 g i j k (ol:('a#'b)list).
   OBJ_GET_ATTR f1 i (OBJ_DEL_LINK f2 g j k ol) = OBJ_GET_ATTR f1 i ol
```

[B-8] OBJ_GET_LINK と OBJ_SET_ATTR の関係

```
OBJ_GET_LINK_OBJ_SET_ATTR
|- !f g i j (a:'c) (ol:('a#'b)list).
   OBJ_GET_LINK f i (OBJ_SET_ATTR g j a ol) = OBJ_GET_LINK f i ol
```

[B-9] OBJ_GET_ATTR と GEN_OBJ の関係

```
OBJ_GET_ATTR_GEN_OBJ
|- !i x (ol:('a#'b)list) (f:'b->'c). let p = GEN_OBJ x ol in
   EXISTS i ol ==> (OBJ_GET_ATTR f i (SND p) = OBJ_GET_ATTR f i ol)
```

[B-10] OBJ_GET_LINK と GEN_OBJ の関係

```
OBJ_GET_LINK_GEN_OBJ
|- !i x (ol:('a#'b)list) f. let p = GEN_OBJ x ol in
   EXISTS i ol ==> (OBJ_GET_LINK f i (SND p) = OBJ_GET_LINK f i ol)
```

[B-11] EXISTS と GEN_OBJ の関係（1）

```
EXISTS_GEN_OBJ_FST_SND
|- !x (ol:('a#'b)list). let p = GEN_OBJ x ol in
   EXISTS (FST p) (SND p) /\ ~(EXISTS (FST p) ol)
```

[B-12] EXISTS と GEN_OBJ の関係（2）

```
EXISTS_GEN_OBJ
|- !i x (ol:('a#'b)list). let p = GEN_OBJ x ol in
   EXISTS i ol ==> EXISTS i (SND p)
```

[B-13] EXISTS と OBJ_SET_ATTR の関係

```
EXISTS_OBJ_SET_ATTR
|- !g i j (a:'c) (ol:('a#'b)list).
   EXISTS i ol = EXISTS i (OBJ_SET_ATTR g j a ol)
```

[B-14] EXISTS と OBJ_INIT_LINK の関係

```
EXISTS_OBJ_INIT_LINK
|- !g i j (ol:('a#'b)list).
   EXISTS i ol = EXISTS i (OBJ_INIT_LINK g j ol)
```

[B-15] EXISTS と OBJ_ADD_LINK の関係

```
EXISTS_OBJ_ADD_LINK
|- !f g i j k (ol:('a#'b)list).
   EXISTS i ol = EXISTS i (OBJ_ADD_LINK f g j k ol)
```

[B-16] EXISTS と OBJ_DEL_LINK の関係

```
EXISTS_OBJ_DEL_LINK
|- !f g i j k (ol:('a#'b)list).
   EXISTS i ol = EXISTS i (OBJ_DEL_LINK f g j k ol)
```

[B-17] OBJ_GET_LINK の性質

```
|- !f i j (ol:('a#'b)list).
   let l = OBJ_GET_LINK f i ol in IS_EL j l ==> (COUNT_EL j l = 1)
```

OBJ_FUN_ATTR に関する定理を以下に示す.

1. OBJ_GET_ATTR と OBJ_FUN_ATTR の関係

```
OBJ_GET_ATTR_OBJ_FUN_ATTR1
|- !(f:'b->'c) g h i (ol:('a#'b)list). EXISTS i ol ==> COND1 f g ==>
   (OBJ_GET_ATTR f i (OBJ_FUN_ATTR f g h i ol) =
    h (OBJ_GET_ATTR f i ol))

OBJ_GET_ATTR_OBJ_FUN_ATTR2
|- !(f1:'b->'c) (f2:'b->'d) g h i (ol:('a#'b)list).
   EXISTS i ol ==> COND2 f1 g ==>
   (OBJ_GET_ATTR f1 i (OBJ_FUN_ATTR f2 g h i ol) =
    OBJ_GET_ATTR f1 i ol)

OBJ_GET_ATTR_OBJ_FUN_ATTR3
|- !(f1:'b->'c) (f2:'b->'d) g h i j (ol:('a#'b)list). ~(i = j) ==>
   (OBJ_GET_ATTR f1 i (OBJ_FUN_ATTR f2 g h j ol) =
    OBJ_GET_ATTR f1 i ol)
```

2. OBJ_GET_LINK と OBJ_FUN_ATTR の関係

```
OBJ_GET_LINK_OBJ_FUN_ATTR
|- !f1 (f2:'b->'c) g h i j (ol:('a#'b)list).
   OBJ_GET_LINK f1 i (OBJ_FUN_ATTR f2 g h j ol) = OBJ_GET_LINK f1 i ol
```

3. EXISTS と OBJ_FUN_ATTR の関係

```
EXISTS_OBJ_FUN_ATTR
|- !(f:'b->'c) g h i j (ol:('a#'b)list).
   EXISTS i ol = EXISTS i (OBJ_FUN_ATTR f g h j ol)
```

第4章　例題

4.1　定理モジュール構築の手順

定理モジュールの構築は以下の手順で行う.

1. オブジェクト指向分析モデルによる領域分析
   (a) 領域分析によりクラスモデルを生成する.
   (b) クラスモデルが決定すると関数を定義するための基本関数, オブジェクト生成関数が与えられる. これを用いて関数を定義する. これは模式言語を利用して行う.
   (c) モデルに制約を与える. これは, 関数に事前条件, 事後条件を与えることによって行う. 条件は参照関数から構成される論理式である.
2. モデルを HOL に実装した公理系に変換する.
3. 模式言語で記述された関数定義を HOL の関数定義に変換する.
4. 制約記述を命題に変換し, 生成された公理系の上で証明する.
5. 他のシステムから利用できるような型を生成する.

4.2　領域分析

例題として, 貸出システム領域を考える. この領域は, 一つのサーバが存在し, 複数の利用者に物品の貸し出しを行うというものである.

ＨＯＬで実装する前に行うことは, クラスモデルによる領域分析である. まず, 領域を構成するクラスとその属性, リンク集合, およびその型を決定する. これが決定すると各クラスの基本関数とオブジェクト生成関数が与えられ, 複合関数が定義できるようになる. 関数の定義は, 可読性を高めるため, 模式言語によって行う.

4.2.1　クラスモデル

クラスモデルを図 4.1 に示す.
以下の 3 つのクラスから構成する.

- 利用者クラス
  - 属性
    - `uid:num`（利用者 ID）

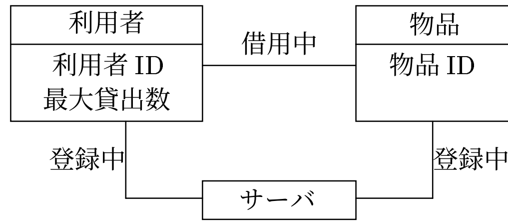

図 4.1: 貸出システムのクラスモデル

    * max:num（最大貸出数）
  - リンク集合
    * il:item list（借用中の物品へのリンク）

- 物品クラス
  - 属性
    * iid:num（物品 ID）
  - リンク集合
    * ul:user list（貸し出している利用者へのリンク）

- サーバクラス
  - リンク集合
    * ul:user list（登録中の利用者へのリンク）
    * il:item list（登録中の物品へのリンク）

4.2.2 関数の定義

まず, オブジェクト生成関数を定義する. 利用者オブジェクト生成関数は gen_user() である. 実装では, オブジェクト生成と同時にそのオブジェクトのコンストラクタ item() を呼び出す new_user() をオブジェクト生成関数として用いる. 同様に new_item,new_srv を定義する.

```
new_item(){let i = gen_item() in {i.item(); return i;}};
new_user(){let u = gen_user() in {u.user(); return u;}};
new_srv(){let s = gen_srv() in {s.srv(); return s;}};
```

各クラスの複合関数の定義を以下に示す.

利用者の複合関数

```
class user{
private:
```

```
    uid:num; // 利用者 ID
    max:num; // 最大貸出数
    il:item list; // 借用中の物品リスト

    // リンク集合操作関数
    get_item_list(){return il;}:item list;
    init_item_list(){il=[];};
    add_item(item i){if(!(IS_EL i il)){il = CONS i il;};}:unit;
    del_item(item i){if(IS_EL i il){il = DEL i il;};}:unit;

public:
    // 属性操作関数
    get_uid(){return uid;}:num;
    set_uid(u:num){uid = u;};
    get_max(){return max;}:num;
    set_max(m:num){max = m;};

    // コンストラクタ.new_user() から呼び出される.
    user(){set_max(5); init_item_list();};

    // 借用数を得る
    get_item_num(){return LENGTH get_item_list();}:num;

    // 引数で指定される物品を借用しているか.
    is_loaning(i:item){return IS_EL i get_item_list();}:bool;

    // 引数で指定される物品を貸出可能か.
    // 同じ物品を借用しておらず, 借用数が最大貸出数以下
    is_loanable(i:item){return !is_loaning(i) &&
                                get_item_num() < get_max()}:bool;

    // 貸出
    loan(i:item){if(is_loanable(i)){add_item(i);};};

    // 返却
    return(i:item){if(is_loaning(i)){del_item(i);};};
};
```

物品の複合関数

```
class item{
private:
```

```
    iid:num;  // 物品 ID
    ul:user list; // 貸し出している利用者リスト

    // リンク集合操作関数
    get_user_list(){return ul;}:user list;
    init_user_list(){ul = [];};
    add_user(u:user){if(~(IS_EL u ul)){ul = CONS u ul;};};
    del_user(u:user){if(IS_EL u ul){ul = DEL u ul;};};

public:
    // 属性操作関数
    get_iid(){return iid;}:num;
    set_iid(i:num){iid = i;};

    // コンストラクタ.new_item() から呼び出される.
    item(){init_user_list();};

private:
    // 貸し出している利用者数を取得.
    get_user_num(){return LENGTH get_user_list();}:num;

public:
    // 貸出可能か. 誰にも貸し出されていなければ貸出可能.
    is_available(){return get_user_num()==0;}:bool;

    // 引数で指定される利用者に貸し出されているか.
    is_loaned_by(u){return IS_EL u get_user_list();}:bool;

    // 貸出
    loan(u){if(is_available()){add_user(u);};};

    // 返却
    return(u){if(is_loaned_by(u){del_user(u);};
};
```

サーバの複合関数

```
class srv{
private:
    ul:user list; // 登録中の利用者リスト
    il:item list; // 登録中の物品リスト
```

```
    // リンク集合操作関数
    get_user_list(){return ul;}:user list;
    get_item_list(){return il;}:item list;
    add_user(u:user){if(~(IS_EL u ul)){ul = CONS u ul;};};
    del_user(u:user){if(IS_EL u ul){ul = DEL u ul;};};
    init_user_list(){ul = [];};
    add_item(i:item){if(~(IS_EL i il)){il = CONS i il;};};
    del_item(i:item){if(IS_EL i il){il = DEL i il;};};
    init_item_list(){il = [];};

public:
    // コンストラクタ.new_srv()から呼び出される.
    srv(){init_user_list(); init_item_list();};

    // 登録中の利用者数を取得
    get_user_num(){return LENGTH get_user_list();}:num;

    // 登録中の物品数を取得
    get_item_num(){return LENGTH get_item_list();}:num;

   // 引数で指定される物品が登録されているか
    item_registered(i:item){return IS_EL i get_item_list();};

    // 引数で指定される利用者が登録されているか
    user_registered(u:user){return IS_EL u get_user_list();};

    // 第一引数の利用者が第二引数の物品を貸出可能か
    is_loanable(u:user,i:item){
        return
            user_registered(u) && // uが登録されている
            item_registered(i) && // iが登録されている
            u.is_loanable(i) &&   // uがiを貸出可能
            i.is_available();     // iが貸出可能
    }:bool;

    // 第一引数の利用者が第二引数の物品を返却可能か
    is_returnable(u:user,i:item){
        return
            user_registered(u) && // uが登録されている
            item_registered(i) && // iが登録されている
            u.is_loaning(i) &&    // uがiを借用中
            i.is_loaned_by(u);    // iがuに貸し出されている
```

```
    }:bool;

    // 利用者の登録
    register_user(){
        let u = new_user() in {add_user(u); return u;};
    }:user;

    // 物品の登録
    register_item(){
        let i = new_item() in {add_item(i); return i;};
    }:item;

    // 物品の削除. 物品が登録されており, 誰にも貸し出されていなければ削除可能.
    delete_item(i:item){
        if(item_registered && i.is_available()){
               del_item(i);
        };
    };

    // 利用者の削除. 利用者が登録されており, 物品を借用していなければ削除可能.
    delete_user(u:user){
        if(user_registered() && u.get_item_num()==0){
               del_user(u);
        };
    };

    // 貸出手続き
    loan(u:user,i:item){
       if(is_loanable(u,i)){
            u.loan(i); i.loan(u)
        };
    };

    // 返却手続き
    return(u:user,i:item){
       if(is_returnable(u,i)){
            u.return(i); i.return(u);
        };
    };

private:
/*  任意のオブジェクトのリストに対し,
```

```
    各オブジェクトにある関数を適用して得られる値の総和をとる関数
    obj_list_sum(f:unit->num,l:'a list){
        if(l==[]){return 0;}
        else{return (HD l).f + obj_list_sum(f,TL l);};
    };
*/
    user_list_get_loan_sum(l:user list){
        return obj_list_sum user_get_item_num l;
    };

    item_list_get_loan_sum(l:item list){
        return obj_list_sum(item_get_user_num,l);
    };
public:
    // 利用者の貸出総数取得
    get_user_loan_sum(){
        return user_list_loan_sum(get_user_list());
    };
    // 物品の貸出総数取得
    get_item_loan_sum(){
        return item_list_loan_sum(get_item_list());
    };

private:
/*  任意のオブジェクトのリストから，
    引数で指定する述語を満たすオブジェクトのリストを返す関数
    obj_list_filter(f:unit->bool,l:'a list){
        if(l==[]){return [];}
        else{if((HD l).f){CONS (HD l) obj_list_filter(P,TL l);}
             else{return obj_list_filter(P,TL l);};}
    };
*/
    item_list_get_available(l:item list){
        return obj_list_filter(is_available,l);
    };
public:
    // 貸出可能な物品数を取得する
    get_available_item_list(){
        return LENGTH item_list_get_available(get_item_list());
    };
};
```

4.2.3 関数の制約記述

関数に対して事前条件, 事後条件を与えることにより, 証明を行う命題を設定する. 条件は参照関数から構成される命題とする. 以下にその一部を示す.

例えば, 利用者クラスの関数 loan() について次の制約を与える.

```
loan(item)
pre  -- get_item_num() = n
post -- get_item_num() = (is_loanable(i) => n + 1 | n)
```

これは利用者が物品を貸し出したときの貸出数の変化を記述するものである. 制約の対象となるのは loan(item) を呼び出した利用者であり, 関数適用前後での参照関数 get_item_num() で得られる値について命題を設定している.

その他の制約例を以下に示す.

- 物品クラスの関数 loan() の制約

  物品が貸出可能の状態で貸し出されると貸出不能となる.

```
loan(user)
pre  -- is_available()
post -- ~is_available()
```

- サーバクラスの関数の制約

  利用者が物品を貸し出したときの利用者の貸出総数の変化

```
loan(user,item)
pre  -- get_user_loan_sum() = n
post -- get_user_loan_sum() = (is_loanable(user,item) => n + 1 | n)
```

- 利用者が物品を貸し出したときの物品の貸出総数の変化

```
loan(user,item)
pre  -- get_item_loan_sum() = n
post -- get_item_loan_sum() = (is_loanable(user,item) => n + 1 | n)
```

- 利用者が物品を貸し出したときの利用者の貸出総数と物品の貸出総数の和は等しい

```
loan(user,item)
pre  -- get_user_loan_sum() = get_item_loan_sum()
post -- get_user_loan_sum() = get_item_loan_sum()
```

4.3 HOL への変換と定理証明

生成したクラスモデルから貸出システム領域の定理モジュールを構築する. 定理モジュール構成を図 4.2 に示す.

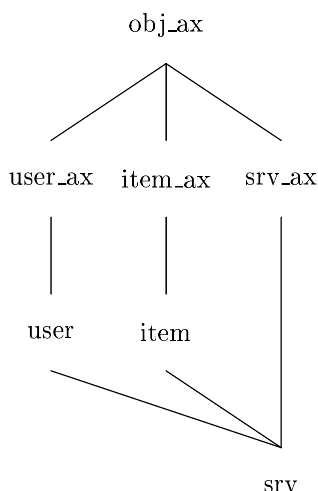

図 4.2: 貸出システムの定理モジュール構成

まず, クラスごとにそのオペレータを含む定理モジュールを生成する. クラスの基本関数, オブジェクト生成関数, 存在検査述語は obj_ax の関数を詳細化することによって得られる. これらの関数, 述語に関する公理群は obj_ax の定理を補題として簡単に生成できる. 各クラスの理論を定理モジュール user_ax,item_ax,srv_ax とする.

次に, 各クラスの複合関数を定義する. 利用者クラスのすべての複合関数は他のクラスの関数（複合関数, オブジェクト生成関数）を呼び出さない. したがって, 利用者クラスの複合関数の定義は, 利用者クラスの基本関数, オブジェクト生成関数のみで定義できる. この定義は,user_ax のみを継承した定理モジュール user で行う. 定義した複合関数に関する定理を証明する. 物品クラスも同様に他のクラスの関数を呼び出さないので,item_ax のみを継承した定理モジュール item を生成し, 複合関数の定義, 定理証明を行う. サーバクラスの複合関数は利用者クラス, 物品クラスの関数を呼び出す. したがって, この定義は, すべてのクラスを含めた領域でなければ行えない. この領域を,user,item,srv_ax を継承する定理モジュール srv とする. この定理モジュールにおいてまず行うことは,user,item,srv_ax に含まれる関数の入出力の拡張である. これらの関数はそれぞれのクラスのオブジェクト集合のみを入出力としており, このままでは領域全体に適用されない. この入出力を領域全体, つまり, すべてのクラスのオブジェクト集合とするように拡張する. サーバクラスの複合関数はこれらの拡張された関数を用いて定義する. そして, 最終的な貸出システム領域の定理を証明する.

4.3.1 user_ax,item_ax,srv_ax

user_ax,item_ax,srv_ax の各定理モジュールでは, 利用者, 物品, サーバの各クラスの公理系を生成する. 生成手順は機械的なものであるため,user_ax についてのみ説明する. 物品, サーバについても同様である.

定義

user_ax では, 利用者クラスの公理系を生成する. 利用者クラスの基本関数の定義は,obj3 で定義された一般的な基本関数の変数を埋めることによって行う. 変数に代入するための関数を以下に定義する.

```
USER_GL1 |- !(t:num list). USER_GL1 t = t
USER_SL1 |- !(s:num list) (t:num list). USER_SL1 s t = s
USER_GA1 |- !(t:num#num). USER_GA1 t = FST t
USER_SA1 |- !(s:num) (t:num#num). USER_SA1 s t = (s,SND t)
USER_GA2 |- !(t:num#num). USER_GA2 t = SND t
USER_SA2 |- !(s:num) (t:num#num). USER_SA2 s t = (FST t,s)
```

- USER_GL1,USER_SL1：利用者オブジェクトの１番目のリンク集合（物品オブジェクト集合）を取得, 更新する.
- USER_GA1,USER_SA1：利用者オブジェクトの１番目の属性（利用者 ID）を取得, 更新する.
- USER_GA2,USER_SA2：利用者オブジェクトの２番目の属性（最大貸出数）を取得, 更新する.

これらの関数について以下の定理が成り立つ.

```
COND1_L1 |- COND1 USER_GL1 USER_SL1
COND1_A1 |- COND1 USER_GA1 USER_SA1
COND1_A2 |- COND1 USER_GA2 USER_SA2
COND2_A1_A2 |- COND2 USER_GA1 USER_SA2
COND2_A2_A1 |- COND2 USER_GA2 USER_SA1
```

- COND1_L1：USER_GL1 と USER_SL1 は同一のリンク集合に対する取得, 更新である.
- COND1_A1：USER_GA1 と USER_SA1 は同一の属性に対する取得, 更新である.
- COND1_A2：USER_GA2 と USER_SA2 は同一の属性に対する取得, 更新である.
- COND2_A1_A2：USER_GA1 と USER_SA2 は異なる属性に対する取得, 更新である.
- COND2_A2_A1：USER_GA2 と USER_SA1 は異なる属性に対する取得, 更新である.

これらの定理は, 基本関数間の定理を obj3 の定理を使って証明する際に必要である.
以上で定義した関数を用いて, 利用者の基本関数を定義する.

- リンク集合操作関数

```
|- !i (ol:(num list#(num#num))list).
   USER_GET_ITEM_LIST1 i ol = OBJ_GET_LINK USER_GL1 i ol

|- !i (ol:(num list#(num#num))list).
   USER_INIT_ITEM_LIST1 i ol = OBJ_INIT_LINK USER_SL1 i ol

|- !i j (ol:(num list#(num#num))list).
   USER_ADD_ITEM1 i j ol = OBJ_ADD_LINK USER_GL1 USER_SL1 i j ol

|- !i j (ol:(num list#(num#num))list).
   USER_DEL_ITEM1 i j ol = OBJ_DEL_LINK USER_GL1 USER_SL1 i j ol
```

- 属性操作関数

```
|- !i (ol:(num list#(num#num))list).
    USER_GET_UID1 i ol = OBJ_GET_ATTR USER_GA1 i ol

|- !i (ol:(num list#(num#num))list).
     USER_SET_UID1 i j ol = OBJ_SET_ATTR USER_SA1 i j ol

|- !i (ol:(num list#(num#num))list).
   USER_GET_MAX1 i ol = OBJ_GET_ATTR USER_GA2 i ol

|- !i j (ol:(num list#(num#num))list).
   USER_SET_MAX1 i j ol = OBJ_SET_ATTR USER_SA2 i j ol
```

オブジェクト生成関数, 存在検査述語の定義を以下に示す.

```
|- !(ol:(num list#(num#num))list). GEN_USER1 ol = GEN_OBJ ol

|- !i (ol:(num list#(num#num))list). USER_EXISTS1 i ol = EXISTS i ol
```

これらの基本関数, オブジェクト生成関数, 存在検査述語に関する定理, つまり, オブジェクト集合の公理は, obj_ax の定理を補題として簡単に証明できる. 証明した定理は付録とする.

4.3.2 user

定義

この定理モジュールは,user_ax を継承し, 利用者クラスの複合関数を定義する. この定義は模式言語の記述をもとに行う.

```
|- !ol.
   NEW_USER1 ol = let p1 = ADD_USER1 ol in
                  let p2 = USER_INIT_ITEM_LIST1 (FST p1) (SND p1) in
                  let p3 = USER_SET_MAX1 (FST p1) MAX p2 in
                  (FST p1,p3)

|- !i ol. USER_GET_ITEM_NUM1 i ol = LENGTH (USER_GET_ITEM_LIST1 i ol)

|- !i ol. USER_IS_LOANING1 i j ol = IS_EL j (USER_GET_ITEM_LIST1 i ol)

|- !i ol. USER_IS_LOANABLE1 i j ol = ~(USER_IS_LOANING1 i j ol) /\
   (USER_GET_ITEM_NUM1 i ol < USER_GET_MAX1 i ol)

|- !i ol. USER_LOAN1 i j ol =
   (USER_IS_LOANABLE1 i j ol => USER_ADD_ITEM1 i j ol | ol)

|- !i ol. USER_RETURN1 i j ol =
   (USER_IS_LOANING1 i j ol => USER_DEL_ITEM1 i j ol | ol)
```

この他に, 基本関数

USER_GET_UID1
USER_SET_UID1
USER_GET_MAX1
USER_SET_MAX1

も利用者クラスの複合関数とする.

定理

定義した関数について定理を証明する. その一部を以下に示す.

- 貸出後の利用者の借用数の変化

```
|- !i j ol. let n = USER_GET_ITEM_NUM1 i ol in EXISTS i ol ==>
   (USER_GET_ITEM_NUM1 i (USER_LOAN1 i j ol) =
   (USER_IS_LOANABLE1 i j ol => n + 1 | n))
```

- 利用者 i が物品 j を貸し出した直後,i は j を貸し出している.

```
|- !i j ol. EXISTS i ol ==> USER_IS_LOANABLE1 i j ol ==>
   USER_IS_LOANING1 i j (USER_LOAN1 i j ol)
```

- 利用者の借用数が最大貸出数と等しいならば, その利用者は貸出不能.

```
|- !i j ol. (USER_GET_ITEM_NUM1 i ol = USER_GET_MAX1 i ol) ==>
   ~(USER_IS_LOANABLE1 i j ol)
```

- 貸出によって利用者の存在の真偽は不変.

```
|- !i j k ol. EXISTS i ol = EXISTS i (USER_LOAN1 j k ol)
```

4.3.3 item

定義

この定理モジュールは,item_ax を継承し, 物品クラスの関数を定義する. この定義は模式言語の記述をもとに行う.

```
|- !i ol. ITEM_GET_USER_NUM1 i ol = LENGTH (ITEM_GET_USER_LIST1 i ol)

|- !i ol. ITEM_IS_AVAILABLE1 i ol = (ITEM_GET_USER_NUM1 i ol = 0)

|- !i j ol. ITEM_IS_LOANED_BY1 i j ol = IS_EL j (ITEM_GET_USER_LIST1 i ol)

|- !i j ol. ITEM_LOAN1 i j ol =
   (ITEM_IS_AVAILABLE1 i ol => ITEM_ADD_USER1 i j ol | ol)

|- !i j ol. ITEM_RETURN1 i j ol =
   (ITEM_IS_LOANED_BY1 i j ol => ITEM_DEL_USER1 i j ol | ol)

|- !ol.
   NEW_ITEM1 ol = let p1 = GEN_ITEM1 ol in
                  let p2 = ITEM_INIT_USER_LIST1 (FST p1) (SND p1) in
                  (FST p1,p2)
```

ITEM_GET_USER_NUM1 は関数ではなく,ITEM_IS_AVAILABLE1 を定義するための補助関数である. この他に, 基本関数

ITEM_GET_IID1
ITEM_SET_IID1

も物品クラスの複合関数とする.

定理

定義した関数について定理を証明する. その一部を以下に示す.

- 物品が貸出可能であることと物品が誰にも貸し出されていないことは同値.

```
|- !i d. ITEM_IS_AVAILABLE1 i d = !j. ~(ITEM_IS_LOANED_BY1 i j d)
```

- 物品が貸し出された直後, その物品は貸出不能.

```
|- !i j ol. EXISTS i ol ==> ~(ITEM_IS_AVAILABLE1 i (ITEM_LOAN1 i j ol))
```

- 物品 i が利用者 j に貸し出された直後,i は j に貸し出されている.

```
|- !i j ol. EXISTS i ol ==> ITEM_IS_AVAILABLE1 i ol ==>
   ITEM_IS_LOANED_BY1 i j (ITEM_LOAN1 i j ol)
```

4.3.4 srv

サーバは利用者、物品の関数を呼び出す. したがって, サーバの関数定義は user,item,srv_ax の 3 つの定理モジュールを継承した定理モジュールにおいて行う. この領域の型は,

```
(num list#(num#num))list)#((num list#num)list)#(((num list#num list)#num)list)
```

である. つまり, 3 つの領域を user,item,srv_ax の順に組としたものである.

領域の拡張

この定理モジュールでは, まず,user,item,srv_ax の関数の入出力を 3 つのオブジェクト集合全体となるように拡張する.

user の関数を拡張するための補助関数を以下に定義する. d は領域であり, 上の型を持つ.

```
GET_USER
|- !d. GET_USER d = FST d
SET_USER
|- !(s:(num list#(num#num))list) d. SET_USER s d = (s,SND d)
USER_FUN
|- !f d. USER_FUN f d = SET_USER (f (GET_USER d)) d
```

- GET_USER：領域の第一要素, つまり, 利用者領域を取得.
- SET_USER：利用者領域を s に更新.
- USER_FUN：利用者領域に関数 f を適用. これは,GET_USER,SET_USER から構成する.

これらの関数を用いて user の関数を定義する. 3 つの関数についてのみ示す. その他の関数についても同様に定義する.

```
USER_GET_UID
|- !i d. USER_GET_UID i d = USER_GET_UID1 i (GET_USER d)
USER_SET_UID
|- !i j d. USER_SET_UID i j d = USER_FUN (USER_SET_UID1 i j) d
NEW_USER
|- !d. NEW_USER d = let d1 = NEW_USER1 (GET_USER d) in
   (FST d1,SET_USER (SND d1) d)
```

- USER_GET_UID（*type*5 の関数）

  GET_USER で取得した利用者領域に対し,user の USER_GET_UID1 を用いて利用者 ID を取得する.

- USER_SET_UID（*type*3 の関数）

  USER_FUN により, 利用者領域に対し,user の USER_SET_UID1 を適用し, 利用者 ID を更新する.

- NEW_USER（*type*2 の関数）

  GET_USER で取得した利用者領域に対し,user の NEW_USER1 を適用する. 得られた利用者オブジェクトはそのまま出力し, 更新された利用者領域は SET_USER を用いて領域に設定する.

item,srv_ax の関数の定義も同様に行う. 拡張のために用いる補助関数を以下に示す.

```
GET_ITEM
|- !d. GET_ITEM d = FST (SND d)
SET_ITEM
|- !(s:((num list#num)list)) d. SET_ITEM s d = (FST d,s,SND (SND d))
ITEM_FUN
|- !f d. ITEM_FUN f d = SET_ITEM (f (GET_ITEM d)) d

GET_SRV
|- !d. GET_SRV d = SND (SND d)
SET_SRV
|- !(s:((num list#num list)#num)list) d.
   SET_SRV s d = (FST d,FST (SND d),s)
SRV_FUN
|- !f d. SRV_FUN f d = SET_SRV (f (GET_SRV d)) d
```

- GET_ITEM：領域の第二要素, つまり, 物品領域を取得.
- SET_ITEM：物品領域を s に更新.
- ITEM_FUN：物品領域に関数 f を適用.
- GET_SRV：領域の第二要素, つまり, サーバ領域を取得.
- SET_SRV：サーバ領域を s に更新.
- SRV_FUN：サーバ領域に関数 f を適用.

サーバの関数定義

サーバの関数を以下に定義する. これは模式言語の記述をもとに行う.

```
|- !i d. SRV_GET_USER_NUM i d = LENGTH (SRV_GET_USER_LIST i d)

|- !i d. SRV_GET_ITEM_NUM i d = LENGTH (SRV_GET_ITEM_LIST i d)

|- !i j d. SRV_USER_REGISTERED i j d = IS_EL j (SRV_GET_USER_LIST i d)

|- !i j d. SRV_ITEM_REGISTERED i j d = IS_EL j (SRV_GET_ITEM_LIST i d)

|- !i user item d. SRV_IS_LOANABLE i user item d =
   SRV_USER_REGISTERED i user d /\ SRV_ITEM_REGISTERED i item d /\
   USER_IS_LOANABLE user item d /\ ITEM_IS_AVAILABLE item d

|- !i user item d. SRV_IS_RETURNABLE i user item d =
   SRV_USER_REGISTERED i user d /\ SRV_ITEM_REGISTERED i item d /\
   USER_IS_LOANING user item d /\ ITEM_IS_LOANED_BY item user d

|- !i d. SRV_REGISTER_USER i d =
   let d1 = NEW_USER d in
   let d2 = SRV_ADD_USER i (FST d1) (SND d1) in (FST d1,d2)

|- !i d. SRV_REGISTER_ITEM i d =
   let d1 = NEW_ITEM d in
   let d2 = SRV_ADD_ITEM i (FST d1) (SND d1) in (FST d1,d2)

|- !i user d. SRV_DELETE_USER i user d =
   (SRV_USER_REGISTERED i user d /\ (USER_GET_ITEM_NUM user d = 0) =>
    SRV_DEL_USER i user d | d)

|- !i item d. SRV_DELETE_ITEM i item d =
   (SRV_ITEM_REGISTERED i item d /\ ITEM_IS_AVAILABLE item d =>
    SRV_DEL_ITEM i item d | d)
```

```
|- !i user item d. SRV_LOAN1 i user item d =
   (SRV_IS_LOANABLE i user item d =>
      let d1 = USER_LOAN user item d in
      let d2 = ITEM_LOAN item user d1 in d2 | d)

|- !i user item d. SRV_RETURN i user item d =
   (SRV_IS_RETURNABLE i user item d =>
      let d1 = USER_RETURN user item d in
      let d2 = ITEM_RETURN item user d in d2 | d)

|- !d.
   NEW_SRV d = let p1 = ADD_SRV d in
               let p2 = SRV_INIT_USER_LIST (FST p1) (SND p1) in
               let p3 = SRV_INIT_ITEM_LIST (FST p1) p2 in (FST p1,p3)

|- !l d. USER_LIST_GET_LOAN_SUM l d = OBJ_LIST_SUM USER_GET_ITEM_NUM l d

|- !i d. SRV_GET_USER_LOAN_SUM i d =
   USER_LIST_GET_LOAN_SUM (SRV_GET_USER_LIST i d) d

|- !l d. ITEM_LIST_GET_LOAN_SUM l d = OBJ_LIST_SUM ITEM_GET_USER_NUM l d

|- !i d. SRV_GET_ITEM_LOAN_SUM i d =
   ITEM_LIST_GET_LOAN_SUM (SRV_GET_ITEM_LIST i d) d

|- !l d. ITEM_LIST_GET_AVAILABLE l d =
   OBJ_LIST_FILTER1 ITEM_IS_AVAILABLE l d

|- !i d. SRV_GET_AVAILABLE_ITEM_LIST i d =
   ITEM_LIST_GET_AVAILABLE (SRV_GET_ITEM_LIST i d) d

|- !i d. SRV_GET_AVAILABLE_ITEM_NUM i d =
   LENGTH (SRV_GET_AVAILABLE_ITEM_LIST i d)
```

4.3.5 貸出システムの定理

貸出システムついて以下の定理を証明した.

- 貸出前後で, 利用者の貸出総数と貸し出されている品物総数は等しい

```
loan(user,item)
pre  -- get_user_loan_sum() = get_item_loan_sum()
post -- get_user_loan_sum() = get_item_loan_sum()

|- !i user item d.
   let d1 = SRV_LOAN1 i user item d in
   SRV_EXISTS i d ==> USER_EXISTS user d ==> ITEM_EXISTS item d ==>
   (SRV_GET_USER_LOAN_SUM i d = SRV_GET_ITEM_LOAN_SUM i d) ==>
   (SRV_GET_USER_LOAN_SUM i d1 = SRV_GET_ITEM_LOAN_SUM i d1)
```

- 利用者を登録後, その利用者は登録されている

```
register_user():user
post -- user_registered(user)

|- !i d. let p = SRV_REGISTER_USER i d in SRV_EXISTS i d ==>
   SRV_USER_REGISTERED i (FST p) (SND p)
```

- 品物を登録後, その品物は登録されている

```
register_item():item
post -- item_registered(item)

|- !i d. let p = SRV_REGISTER_ITEM i d in SRV_EXISTS i d ==>
   SRV_ITEM_REGISTERED i (FST p) (SND p)
```

- 利用者登録前後の登録利用者総数の変化

```
register_user():user
pre  -- ~user_registered(user)
post -- get_user_num() = get_user_num()@pre + 1

|- !i d. let p = SRV_REGISTER_USER i d in SRV_EXISTS i d ==>
   ~(SRV_USER_REGISTERED i (FST p) d) ==>
   (SRV_GET_USER_NUM i (SND p) = SRV_GET_USER_NUM i d + 1)
```

- 品物登録前後の登録利用者総数の変化

```
register_item():item
pre  -- ~item_registered(item)
post -- get_item_num() = get_item_num()@pre + 1

|- !i d. let p = SRV_REGISTER_ITEM i d in SRV_EXISTS i d ==>
   ~(SRV_ITEM_REGISTERED i (FST p) d) ==>
   (SRV_GET_ITEM_NUM i (SND p) = SRV_GET_ITEM_NUM i d + 1
```

- 利用者Aが品物Bを貸し出したとき, BはAに貸し出されている

```
|- !i item user d. let d1 = SRV_LOAN1 i user item d in
   ITEM_EXISTS item d ==>
   SRV_IS_LOANABLE i user item d ==> ITEM_IS_LOANED_BY item user d1
```

- 品物が貸し出されると, その品物は貸出不能

```
|- !i user item d. let d1 = SRV_LOAN1 i user item d in
   ITEM_EXISTS item d ==>
   SRV_IS_LOANABLE i user item d ==> ~(ITEM_IS_AVAILABLE item d1)
```

- 品物が登録された直後, その品物は貸出可能

```
|- !i d. let d1 = SRV_REGISTER_ITEM i d in
   ITEM_IS_AVAILABLE (FST d1) (SND d1)
```

- 品物登録前後の貸出可能な品物総数は 1 増加する

```
|- SRV_COND i d =
   !j. IS_EL j (SRV_GET_ITEM_LIST i d) ==> ITEM_EXISTS j d

|- !i d. let d1 = SRV_REGISTER_ITEM i d in
   SRV_EXISTS i d ==> SRV_COND i d ==>
   (SRV_GET_AVAILABLE_ITEM_NUM i (SND d1) =
    SRV_GET_AVAILABLE_ITEM_NUM i d + 1)
```

4.4 型の生成

得られた貸出システム領域の定理モジュール srv に含まれる関数は入出力としてオブジェクトのポインタと領域を持つ. また, 定理にはオブジェクトの存在に関する前提条件が含まれている. このようなオブジェ

クト指向のメカニズムが関数, 定理に現れている状態では, F-Developer 等のシステムから使うことができない.

ここでは露呈されているオブジェクト指向のメカニズムを, 型を生成することによって隠蔽する. そして最終的に他のシステムから利用できる定理モジュールを得る.

貸出システム領域には, 利用者, 物品, サーバの３つのクラスが存在し, これらを型とした.

ＨＯＬでは, 新しい型を生成する際, 既存の型の部分集合によりその型を表現する. 簡単に述べると, 次の手続きにより型を生成する.

1. 既存の型を指定する.
2. その型の部分集合を決定する特徴述語を定義する.
3. この部分集合が少なくとも一つ要素をもつことを証明する.
4. 新しい型を定義し, その型から部分集合への全単射を定義する.

4.4.1 サーバ型の生成

以下に, 貸出システム領域の型から, サーバ型 `srv` を生成する手続きを示す. これは定理モジュール srv を継承する定理モジュール srv_type において行う.

1. `srv` 型を表現する型は, 貸出システム領域の型, つまり,

   ```
   (num list#(num#num))list#(num list#num)list#((num list#num list)#num)list
   ```

   とする.

2. 領域の型の要素として, サーバオブジェクト集合にサーバを一つ生成した集合 `SRV_INIT_REP` を定義する.

   ```
   SRV_INIT_REP = SND (NEW_SRV1 ([],[],[]))
   ```

   領域の型について,`srv` 型を表現する部分集合を定める特徴述語を以下に定義する.

   ```
   IS_SRV_REP
   |- !d. IS_SRV_REP d =
             (!P. P SRV_INIT_REP /\
                  !d. P d ==> P (SND (SRV_REGISTER_USER 0 d)) /\
                  !d. P d ==> P (SND (SRV_REGISTER_ITEM 0 d)) /\
                  !d i. P d ==> P (SRV_DELETE_USER 0 i d) /\
                  !d i. P d ==> P (SRV_DELETE_ITEM 0 i d) /\
                  !d i j. P d ==> P (SRV_LOAN 0 i j d) /\
                  !d i j. P d ==> P (SRV_RETURN 0 i j d)) ==> P d
   ```

   `IS_SRV_REP d` は, 領域の型の要素 `d` が `srv` 型を表現する要素であることを意味する. この述語を満たすのは `SRV_INIT_REP` と, それに対して

SRV_REGISTER_USER 0
SRV_REGISTER_ITEM 0
SRV_DELETE_USER 0 i
SRV_DELETE_ITEM 0 i
SRV_LOAN 0 i j
SRV_RETURN 0 i j

を再帰的に適用して得られる領域の集合である．関数の第一引数を 0 とすることにより, 領域に一つ存在しているサーバに関数を適用している．

3. 上の特徴述語で決定される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. この要素は SRV_INIT_REP である．

```
|- ?l. IS_SRV_REP l
```

この定理は,IS_SRV_REP の定義で書き換え,SRV_INIT_REP で l を置き換えると即座に証明できる．

4. ML 関数 new_type_definition によって,srv 型を生成する．

```
> val user_TY_DEF = new_type_definition
     {name = "srv",
      pred = ``IS_SRV_REP``,
      inhab_thm = EXISTS_SRV_REP};;
```

new_type_definition は srv 型と上で定義した部分集合の間に全単射が存在することを表明する. 実際にその存在する全単射を ML 関数 define_new_type_bijection によって生成する．

```
> val srv_ISO_DEF = define_new_type_bijections
                     {name = "srv_ISO_DEF",
                      ABS = "ABS_srv",
                      REP = "REP_srv",
                      tyax = srv_TY_DEF};;
```

ABS_srv は srv から部分集合への全単射,REP_srv は部分集合から srv への全単射である．また, この２つの全単射の性質を以下の ML 関数により得ることができる．

```
> val R_11   = prove_rep_fn_one_one srv_ISO_DEF;;
> val R_ONTO = prove_rep_fn_onto srv_ISO_DEF;;
> val A_11   = prove_abs_fn_one_one srv_ISO_DEF;;
> val A_ONTO = prove_abs_fn_onto srv_ISO_DEF;;
```

- R_11：srv が単射である．
- R_ONTO：srv が全射である．
- A_11：ABS_srv が単射である．

- A_ONTO：ABS_srv が全射である.

5. 生成された srv 型について定数, オペレータを導入する.

   まず,srv 型に定数 SRV を導入する.SRV はサーバの初期値であり, 領域の型の要素 SRV_INIT_REP が表現する.

```
|- SRV_INIT = ABS_srv SRV_INIT_REP
```

   次に,srv 型にオペレータを導入する. 次の関数は, サーバに利用者を登録する関数である.

```
|- !(s:srv). REGISTER_USER s =
   let d = SRV_REGISTER_USER 0 (REP_srv s) in (FST d,ABS_srv (SND d))
```

   まず,srv 型の引数 s を REP_srv によって領域の型へ写す. この値に対して,SRV_REGISTER_USER を適用する. 得られた出力の第一要素, つまり, 登録された利用者はそのまま出力し, 出力の第二要素, つまり, 適用後の領域は,ABS_srv によって,srv 型に戻して出力する.

   その他の関数も同様に定義する.

```
|- !s. REGISTER_ITEM s = let d = SRV_REGISTER_ITEM 0 (REP_srv s) in
                         (FST d,ABS_srv(SND d))
|- !i s. DELETE_USER i s = ABS_srv (SRV_DELETE_USER 0 i (REP_srv s))

|- !i s. DELETE_ITEM i s = ABS_srv (SRV_DELETE_ITEM 0 i (REP_srv s))

|- !i j s. LOAN i j s = ABS_srv (SRV_LOAN 0 i j (REP_srv s))

|- !i j s. RETURN i j s = ABS_srv (SRV_RETURN 0 i j (REP_srv s))

|- !s. GET_USER_NUM s = SRV_GET_USER_NUM 0 (REP_srv s)

|- !s. GET_ITEM_NUM s = SRV_GET_ITEM_NUM 0 (REP_srv s)

|- !i s. USER_REGISTERED i s = SRV_USER_REGISTERED 0 i (REP_srv s)

|- !i s. ITEM_REGISTERED i s = SRV_ITEM_REGISTERED 0 i (REP_srv s)

|- !i j s. IS_LOANABLE i j s = SRV_IS_LOANABLE 0 i j (REP_srv s)

|- !i j s. IS_RETURNABLE i j s = SRV_IS_RETURNABLE 0 i j (REP_srv s)
```

6. サーバクラスの関数について証明した定理をもとに srv 型の公理を生成する. 以下にその一部を示す.

   - 貸出前後の利用者の貸出総数の変化

```
|- !s user item.
   GET_USER_LOAN_SUM (LOAN user item s) =
   (IS_LOANABLE user item s => GET_USER_LOAN_SUM i s + 1 |
                               GET_USER_LOAN_SUM i s)
```

- 貸出前後の貸し出されている品物総数の変化

```
|- !s user item.
   GET_ITEM_LOAN_SUM (LOAN user item s) =
   (IS_LOANABLE user item s => GET_ITEM_LOAN_SUM s + 1 |
                               GET_ITEM_LOAN_SUM s)
```

- 貸出前後で, 利用者の貸出総数と貸し出されている品物総数は等しい

```
|- !s user item.
   let s1 = LOAN user item s in
   (GET_USER_LOAN_SUM s = GET_ITEM_LOAN_SUM s) ==>
   (GET_USER_LOAN_SUM s1 = GET_ITEM_LOAN_SUM s1)
```

- 利用者を登録後, その利用者は登録されている

```
|- !s. let s1 = REGISTER_USER s in
   USER_REGISTERED (FST s1) (SND s1)
```

- 品物を登録後, その品物は登録されている

```
|- !s. let s1 = REGISTER_ITEM s in
   ITEM_REGISTERED (FST s1) (SND s1)
```

- 利用者登録前後の登録利用者総数の変化

```
|- !i s. let s1 = REGISTER_USER i s in
   (GET_USER_NUM (SND s1) = GET_USER_NUM s + 1)
```

- 品物登録前後の登録利用者総数の変化

```
|- !s. let s1 = REGISTER_ITEM s in
   (GET_ITEM_NUM (SND s1) = GET_ITEM_NUM s1 + 1
```

- 品物登録前後の貸出可能な品物総数の変化

```
|- !s. let s1 = REGISTER_ITEM s in
    (GET_AVAILABLE_ITEM_NUM (SND s1) = GET_AVAILABLE_ITEM_NUM s + 1)
```

- 最初, サーバの登録利用者数は 0

```
|- GET_USER_NUM SRV_INIT = 0
```

- 最初, サーバの登録物品数は 0

```
|- GET_ITEM_NUM SRV_INIT = 0
```

- 最初, サーバの貸出可能物品数は 0

```
|- GET_AVAILABLE_ITEM_NUM SRV_INIT = 0
```

4.4.2 利用者型の生成

利用者型 `user` も `srv` 型と同様に領域の型から生成することができるが, 利用者クラスは定理モジュール user として独立しているので, 利用者オブジェクト集合の型

```
(num list#(num#num))list
```

から生成する.`srv` 型と同様の手順で生成する. これは定理モジュール user を継承する定理モジュール user_type において行う. 以下 `user` 型に導入される定数, オペレータ, その公理のみを示す.

- 定数

```
USER_INIT:user
```

- オペレータ

```
GET_UID     :user->num
SET_UID     :num->user
GET_MAX     :user->num
SET_MAX     :num->user
GET_ITEM_NUM:user->num
IS_LOANING  :num->user->bool
IS_LOANABLE :num->user->bool
LOAN        :num->user->user
RETURN      :num->user->user
```

- 公理

```
|- GET_ITEM_NUM USER_INIT = 0
|- !i. IS_LOANABLE i USER_INIT
|- !i. ~(IS_LOANING i USER_INIT)
|- !i u. GET_MAX (SET_UID i u) = GET_MAX u
|- !i u. GET_MAX (SET_MAX i u) = i
```

```
|- !i u. GET_MAX (LOAN i u) = GET_MAX u
|- !i u. GET_MAX (RETURN i u) = GET_MAX u
|- !i u. GET_UID (SET_UID i u) = i
|- !i u. GET_UID (SET_MAX i u) = GET_UID u
|- !i u. GET_UID (LOAN i u) = GET_UID u
|- !i u. GET_UID (RETURN i u) = GET_UID u
|- !i u. let n = GET_ITEM_NUM u in
   (GET_ITEM_NUM (LOAN i u) = (IS_LOANABLE i u => n + 1 | n))
|- !i u. IS_LOANABLE i u ==> ~(USER_IS_LOANING1 i u)
|- !i u. IS_LOANABLE i u ==> GET_ITEM_NUM u < GET_MAX u
|- !i u. IS_LOANABLE i u ==> IS_LOANING i (LOAN i u)
|- !i u. IS_LOANING i u ==> ~(IS_LOANABLE i u)
|- !i u. (GET_ITEM_NUM u = GET_MAX u) ==> ~(IS_LOANABLE i u)
|- !i j u. IS_LOANING i u ==> IS_LOANING i (LOAN j u )
```

第5章　考察

5.1　証明例

以下に示すのは, 利用者の貸出前後での貸出数の変化を述べた命題である. これを Goal Oriented Proof により証明する.

```
     Initial goal:
     ``!i user item d.
         let d1 = SRV_LOAN i user item d in
         let n = USER_GET_ITEM_NUM user d in
         USER_EXISTS user d ==>
         SRV_IS_LOANABLE i user item d ==>
         (USER_GET_ITEM_NUM user d1 = n + 1)``
 : proofs
```

まず, 複合関数である SRV_LOAN を, その定義で書き換える. 同時に,let の展開, 全称限定子, 前提条件を分解する.

```
> e (REWRITE_TAC [SRV_LOAN] THEN
     CONV_TAC (DEPTH_CONV let_CONV) THEN
     REPEAT STRIP_TAC);;

- - OK..
1 subgoal:
val it =
  ``USER_GET_ITEM_NUM user
      ((SRV_IS_LOANABLE i user item d)
       => (ITEM_LOAN item user (USER_LOAN user item d))
       | d) =
    USER_GET_ITEM_NUM user d + 1``
  ----------------------------
      ``USER_EXISTS user d``
      ``SRV_IS_LOANABLE i user item d``
   : goalstack
```

仮定 SRV_IS_LOANABLE i user item d により, 条件項の展開を行う.

```
> e (ASM_REWRITE_TAC []);;

1 subgoal:
val it =
  ``USER_GET_ITEM_NUM user (ITEM_LOAN item user (USER_LOAN user item d)) =
    USER_GET_ITEM_NUM user d + 1``
  ____________________________
      ``USER_EXISTS user d``
      ``SRV_IS_LOANABLE i user item d``
   : goalstack
```

USER_GET_ITEM_NUM と ITEM_LOAN に注目する. この二つの関数の間には, 以下の定理が成り立つ.

```
USER_GET_ITEM_NUM_ITEM_LOAN
|- !i j k d. USER_GET_ITEM_NUM i (ITEM_LOAN j k d) = USER_GET_ITEM_LOAN i d
```

ITEM_LOAN は物品オブジェクト j の値のみを変更する. よって,ITEM_LOAN が適用された直後, 利用者オブジェクト i が USER_GET_ITEM_LIST によってそのリンク集合を取得したとき, その値は,ITEM_LOAN が適用される前に取得する値と同じである. これは複合関数 USRE_GET_ITEM_NUM,ITEM_LOAN をその定義で分解すれば即座に証明される. この定理で書き換えを行う.

```
> e (REWRITE_TAC [USER_GET_ITEM_NUM_ITEM_LOAN]);;

- - OK..
1 subgoal:
val it =
  ``USER_GET_ITEM_NUM user (USER_LOAN user item d) =
    USER_GET_ITEM_NUM user d + 1``
  ____________________________
      ``USER_EXISTS user d``
      ``SRV_IS_LOANABLE i user item d``
   : goalstack
```

USER_GET_ITEM_LIST と USER_LOAN に注目する. この二つの関数の間には以下の定理が成り立つ.

```
USER_GET_ITEM_NUM_USER_LOAN
|- !i j d. let n = USER_GET_ITEM_NUM i d in USER_EXISTS i d ==>
   (USER_GET_ITEM_NUM i (USER_LOAN i j d) =
        ((USER_IS_LOANABLE i j d) => n + 1 | n))
```

これは, 利用者オブジェクト i に対する USER_LOAN の適用前後での, 同じオブジェクトの USER_GET_ITEM_LIST で取得する値の変化を述べた定理である. これは複合関数 USER_GET_ITEM_NUM,USER_LOAN をその定義で分解し, 定理モジュール user に含まれる定理を補題とすることにより証明される.

この定理で書き換えを行うためには,USER_EXISTS user d という条件が必要である. この条件はゴール

の仮定に含まれているので, その仮定とモーダス・ポーネンズを適用する. さらに, 新たに生成された仮定で書き換えを行う.

```
> e (IMP_RES_TAC (REWRITE_RULE (CONV_RULE (DEPTH_CONV let_CONV)
                   USER_GET_ITEM_NUM_USER_LOAN)) THEN
     ASM_REWRITE_TAC []);;

- - OK..
1 subgoal:
val it =
  ``((USER_IS_LOANABLE user item d)
     => (USER_GET_ITEM_NUM user d) + 1
     | (USER_GET_ITEM_NUM user d)) =
    (USER_GET_ITEM_NUM user d) + 1``
  ____________________________
      ``USER_EXISTS user d``
      ``SRV_IS_LOANABLE i user item d``
      ``!j.
          USER_GET_ITEM_NUM user (USER_LOAN user j d) =
          ((USER_IS_LOANABLE user j d)
           => (USER_GET_ITEM_NUM user d) + 1
           | (USER_GET_ITEM_NUM user d))``
   : goalstack
```

条件項の条件 USER_IS_LOANABLE user item d が成り立てばゴールが証明できると分かる. 仮定リストに含まれる SRV_IS_LOANABLE に関して以下の定理が存在する. これは SRV_IS_LOANABLE の定義から即座に導かれる.

```
SRV_IS_LOANABLE_IMP_USER_IS_LOANABLE
|- !i user item d.
   SRV_IS_LOANABLE i user item d ==> USER_IS_LOANABLE user item d
```

この定理とゴールの仮定でモーダス・ポーネンズを適用する.

```
> e (IMP_RES_TAC SRV_IS_LOANABLE_IMP_USER_IS_LOANABLE);;

- - OK..
1 subgoal:
val it =
  ``((USER_IS_LOANABLE user item d)
     => (CONS item (USER_GET_ITEM_NUM user d))
     | (USER_GET_ITEM_NUM user d)) =
    (USER_GET_ITEM_NUM user d) + 1``
  ____________________________
      ``USER_EXISTS user d``
      ``SRV_IS_LOANABLE i user item d``
      ``!j.
          USER_GET_ITEM_NUM user (USER_LOAN user j d) =
          ((USER_IS_LOANABLE user j d)
           => (USER_GET_ITEM_NUM user d) + 1
           | (USER_GET_ITEM_NUM user d))``
      ``USER_IS_LOANABLE user item d``
   : goalstack
```

最後に, 仮定によって条件項を分解する.

```
> e (ASM_REWRITE_TAC []);;

- - OK..
  Initial goal proved.
  |- !i user item d.
       let d1 = SRV_LOAN i user item d
       in
       let n = USER_GET_ITEM_NUM user d
       in
       USER_EXISTS user d ==>
       SRV_IS_LOANABLE i user item d ==>
       (USER_GET_ITEM_NUM user d1 = n + 1) : goalstack
```

5.2 帰納法の回避

帰納法による証明は一般に複雑になる. 帰納法を避ける方法として, 予め, 典型的な再帰的関数を用意しておき, それに関する定理を証明しておく方法がある.

例えば, 頻繁に行われる操作として, リストに含まれるすべてのオブジェクトに対してある関数を適用し, その値の総和を取るというものがある. 次に示す関数は, この操作を行う一般的な関数である.

```
OBJ_LIST_SUM
|- (!f d. OBJ_LIST_SUM f [] d = 0) /\
   (!f x l d. OBJ_LIST_SUM f (CONS x l) d = f x d + OBJ_LIST_SUM f l d)
```

OBJ_LIST_SUM f l d は,l に含まれるオブジェクトに対して, 順番に,f という関数を適用してある値を取得し, その和をとっていく.

この関数に関して次の定理を証明することができる.

```
OBJ_LIST_SUM_ADD1
|- !f g i d l.
   (COUNT_EL i l = 1) ==>
   (f i (g d) = f i d + 1) ==>
   (!j. ~(j = i) ==> (f j (g d) = f j d)) ==>
   (OBJ_LIST_SUM f l (g d) = OBJ_LIST_SUM f l d + 1)
```

この定理の意味は次のようになる. いま, オブジェクト i がリスト l に一つだけ存在する（前提条件の一つ目）. この i に f を適用して得られる値が, 関数 g の適用後に 1 増加する（前提条件の二つ目）. 一方,i とは異なるすべてのオブジェクトに対し,f を適用して得られる値が,g の適用前後で等しい（前提条件の三つ目）. 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 1 増加する（図 5.1）. この定理は, リスト構造に関する帰納法によって証明することができる.

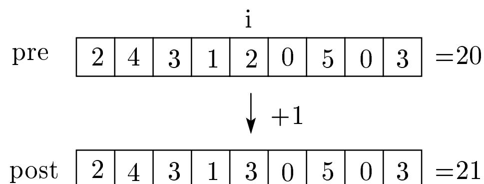

図 5.1: OBJ_LIST_SUM_ADD1 の意味

このような操作を行う具体的な関数を定義する際は,OBJ_LIST_SUM の関数変数 f に具体的な関数を代入すればよい. そうすれば, 上の定理を, 3 つの前提条件を証明するだけで, 書き換え規則として用いることができる.

次の命題は, サーバが貸出手続き SRV_LOAN を適用する前後での, 登録されているすべての利用者の貸出総数の変化を述べたものである.

```
g `!i user item d.
      let d1 = SRV_LOAN i user item d in
  SRV_EXISTS i d ==> USER_EXISTS user d ==>
      SRV_IS_LOANABLE i user item d ==>
      (SRV_GET_USER_LOAN_SUM i d1 =
       SRV_GET_USER_LOAN_SUM i d + 1)`;;
```

これを Tactic によって分解していくと, 次のゴールを得る.

```
- - OK..
1 subgoal:
  ``USER_LIST_GET_LOAN_SUM (SRV_GET_USER_LIST i d) (USER_LOAN user item d) =
    USER_LIST_GET_LOAN_SUM (SRV_GET_USER_LIST i d) d + 1``
  ----------------------------
      ``SRV_EXISTS i d``
      ``USER_EXISTS user d``
      ``SRV_IS_LOANABLE i user item d``
   : goalstack
```

USER_LIST_GET_LOAN_SUM l d は,l に含まれるすべての利用者オブジェクトの貸出総数を求める関数である. この関数は,OBJ_LIST_SUM を用いて, 次のように定義されている.

```
|- !l d. USER_LIST_GET_LOAN_SUM l d = OBJ_LIST_SUM USER_GET_ITEM_NUM l d
```

USER_GET_ITEM_NUM i d は, 利用者 i の貸出数を取得する関数である. 上のゴールをこの定義で書き換える.

```
- - OK..
1 subgoal:
val it =
  ``OBJ_LIST_SUM USER_GET_ITEM_NUM (SRV_GET_USER_LIST i d)
      (USER_LOAN user item d) =
    OBJ_LIST_SUM USER_GET_ITEM_NUM (SRV_GET_USER_LIST i d) d + 1``
  ----------------------------
      ``SRV_EXISTS i d``
      ``USER_EXISTS user d``
      ``SRV_IS_LOANABLE i user item d``
   : goalstack
```

これを証明するためには, 予め証明された OBJ_LIST_SUM_ADD1 を用いればよい. OBJ_LIST_SUM_ADD1 の型を INST_TYPE で適切な型に直し,SPECL により, 適切な値に詳細化すると以下の定理を得る.

```
|- (COUNT_EL user (SRV_GET_USER_LIST i d) = 1) ==>
   (USER_GET_ITEM_NUM user (USER_LOAN user item d) =
    USER_GET_ITEM_NUM user d + 1) ==>
   (!j. ~(j = user) ==>
    (USER_GET_ITEM_NUM j (USER_LOAN user item d) =
     USER_GET_ITEM_NUM j d)) ==>
   (OBJ_LIST_SUM USER_GET_ITEM_NUM (SRV_GET_USER_LIST i d)
    (USER_LOAN user item d) =
    OBJ_LIST_SUM USER_GET_ITEM_NUM (SRV_GET_USER_LIST i d) d + 1)
```

この定理の 3 つの前提条件を別に証明し, モーダス・ポーネンズを適用することにより, ゴールを証明することができる. 一つ目の前提は, ゴールの仮定と, 基本関数 SRV_GET_USER_LIST の以下の定理から証明できる.

```
|- let l = SRV_GET_USER_LIST i d in IS_EL j l ==> (COUNT_EL j l = 1)
```

この定理は, リンク集合には要素が重複して含まれないことを示す定理である. 二つ目, 三つ目の前提は, 複合関数をその定義で分解し, 基本関数間の定理を用いることにより証明できる.

OBJ_LIST_SUM のような典型的な操作を行う関数は, その定理とともに, 定理モジュール util に格納されている.

5.3 命題の設定

命題は, 関数により記述する. その命題は領域の公理系によって証明することになるが, あらゆる命題がその公理系から証明されるとは限らない. 人間が直感的に考えて正しそうな命題も, その公理系が表現できない世界の命題は証明することはできない. 例えば,「生成されたオブジェクトはいつか必ず参照される」といった命題は, 公理系に時間の概念が存在しないために証明することはできない.

領域の公理系は基本関数, オブジェクト生成関数, 存在検査述語から構成されており, それが表現するのは, 関数が適用された前後での領域の状態の変化である. したがって, 複合関数に関して設定する命題は, 関数が適用された前後での領域の値の変化を記述するものでなければならない. 具体的な命題の形式を以下に述べる.

複合関数は,$type1, ..., type6$ の 6 種類の型を持つ. $type1, ..., type4$ の関数は領域の状態を変化させる関数であり, 状態変化関数を呼ぶ.$type5, type6$ の関数は領域の状態を参照する関数であり, 参照関数と呼ぶ. 領域中に存在する状態変化関数 f（$type3$ とする）と参照関数 g（$type6$ とする）について, 以下の命題を証明する. 命題によっては前提条件を伴うこともある.

```
g o1 (f o2 a d) = h (g o1 d)
```

これは,f の適用前後での g で取得される値の変化を記述するものである. h は値の変化を表す. f,g が同じクラスの関数であれば,o1 と o2 が同一のオブジェクトのときと異なるオブジェクトのときの 2 つの場合を証明する. また, 存在検査述語（$type6$）についても同様に以下を証明する.

```
CLASS_EXISTS o1 = CLASS_EXISTS o2 (f d)
```

これは,f の適用前後でオブジェクト o の存在の真偽は不変であることを示す.

基本的には以上のような, 状態変化関数の適用前後での参照関数により取得される値の変化を記述する命題を設定すればよい. このような命題は, 複合関数の定義で書き換えを行うと, 基本関数に関する命題となり, 領域の公理系を用いて証明することができる.

実際はこれ以外の命題も証明することができる. 例えば, 関数の適用前後によらず, トートロジーとして証明される定理などである. どのような命題を設定すればよいかについては, 領域に依存する部分もあり, 考察の余地がある.

5.4 NULL ポインタ

本研究で構築したオブジェクト指向の理論では, オブジェクトのデータはポインタで参照している. ポインタによって参照されるオブジェクトが必ずしもオブジェクト集合に存在するとは限らない. つまり, ポインタが NULL である可能性がある. 多くの定理は, 次の定理のように, ポインタが NULL でないことが前提となっている. 他のシステムから利用できるような型を生成するときはそのようなポインタに関する前提条件は命題に含まれているべきではない.

```
|- !i user item d.
   let d1 = SRV_LOAN1 i user item d in
   SRV_EXISTS i d ==> USER_EXISTS user d ==> ITEM_EXISTS item d ==>
   (SRV_GET_USER_LOAN_SUM i d = SRV_GET_ITEM_LOAN_SUM i d) ==>
   (SRV_GET_USER_LOAN_SUM i d1 = SRV_GET_ITEM_LOAN_SUM i d1)
```

多くの場合, オブジェクトはリンク集合から取得され, 参照される. したがって,「リンク集合に含まれるオブジェクトは必ず領域に存在する」ことを証明すれば多くの定理のポインタに関する前提条件を消すことができる. しかし, この命題は領域の公理系の表現範囲を超えており, 証明することができない.

しかし, 型を生成するときに定義する特徴述語を用いれば証明することができる. サーバ型を生成するときに定義した特徴述語 IS_SRV_REP は,「すべての述語は, 領域の初期値と, それに対して状態を変化させる関数を再帰的に定義して得られる領域について成り立てば, 特徴述語で定められる部分集合のすべての要素に対して成り立つ」ことを示している. したがって,「リンク集合に含まれるオブジェクトは必ず領域に存在する」という述語を, 領域の初期値と, それに対して関数を再帰的に適用して得られる領域について成り立つことを証明すればすべての領域について成り立つことを証明できる. 上の定理では USER_EXISTS user d と ITEM_EXISTS item d をこの方法で消すことができる.SRV_EXISTS i d は srv 型を生成する際の領域の初期値としてサーバを一つ存在させ, そのポインタ 0 で参照するので証明することができる.

ただし, 必ずしも参照されるオブジェクトすべてがリンク集合から取得されるものではないのですべての定理について前提条件を消すことができるわけではない. これを避けるには, 命題の設定の仕方に制限を与え,NULL ポインタが参照されないようにする方法が考えられる.

5.5 定理モジュールの再利用性

利用者や物品クラスはその関数が他のクラスの関数を呼び出さないので独立した定理モジュールで構成することができる. サーバのように他のクラスの関数を呼び出すクラスは一つの定理モジュールで構成することはできない. 他のクラスの関数を呼び出す関数を定義する際は, その 2 つのクラスの定理モジュールを親とする一つの定理モジュールを生成し, その中で関数を定義しなければならない.

このように, 関数呼出しが閉じる範囲で一つの定理モジュールを生成することができる. したがって, 領域において関数呼出しが閉じる範囲を再利用の単位と考えることできる. 利用者や物品のように関数呼出しがそのクラスで閉じるクラスは,「クラス」という単位で定理モジュールを生成できるので部品としての再利用性が高い. しかし, これは特別な場合であり, 本研究の手法では一般にクラスは他のクラスの関数を呼び出すので, 一つのクラスとして定理モジュールを生成することはできない.「関数呼出しが閉じる範囲」というのは人間が考える上で扱いやすい単位ではなく, このような定理モジュールは再利用性が低い. したがって, 本研究の手法では領域全体として一つの定理モジュールは生成することができるが, 領域から部分的に再利用可能な要素を抽出するのは難しい.

本来は, 領域に存在する一つ一つの要素を独立した定理モジュールとして持つ再利用性の高い領域ライブラリを生成すべきであったが, 本研究の手法の現状ではそのような再利用性の高いライブラリを生成することができず, 課題の残る結果となってしまった.

第6章　今後の課題とまとめ

6.1　今後の課題

自動化

関数定義, 命題の設定を HOL で直接行うのは可読性に欠け, 間違いの元になる. これらを行うための言語を設計し HOL に変換する仕組みを生成することが必要である. また,obj3 以降, 領域の理論を得るまでは証明がパターン化しており, 自動で行うことができると考えられる. このパターンを明らかにし,ML 等で実装する必要がある.

定理モジュールの利用法

本研究で生成した利用者型や物品型は, 検証システムから直接, 利用できると考えられる. 一方, サーバ型は, サーバーシステムが検証対象のシステムの一部となっている場合はともかく, サーバシステム自体が検証対象になっているときは利用できない. このように, 型の表す領域が広くなるほどその型は検証システムからの利用可能性が低下してしまう. しかし, サーバー型を作る際に生成したクラスモデルは関数の振る舞いが定義されており, その定理が証明されている. この定理が証明されたサーバーのクラスモデルに変更を加えていくことにより, 実際のシステム, 例えば図書館システムなどのクラスモデルを得ることができる. これを行うためにはサーバーを一般的な形で定義し, システムに応じて詳細化できるような仕組みが必要である.

6.2　まとめ

本研究では, あらゆる領域に適用可能な定理モジュールの構築法を提案した. この手法では, あらゆる領域に適用可能とするために, オブジェクト指向分析モデルを基礎として定理モジュールを構築する. また, オブジェクト指向分析モデルは人間の思考の自然な表現であり, このモデルから定理モジュールを生成することにより, 思考の単位となるような定理を得ることが可能である.

まず, 領域の分析モデルを定義し, それに対応する領域の理論を策定した. 領域は個々のオブジェクトの属性と, オブジェクト間に張られているリンクによって表現する. 領域の分析モデルとして, クラスモデルを導入した. クラスは, 属性, リンク集合, 関数を持つ.

領域の理論をＨＯＬの既存の理論から実装した. まず, 一般的なクラスのオペレータの定義と定理を格納した定理モジュール obj_ax を構築した. 特定の領域が与えられたとき,obj_ax のオペレータをそのクラスの情報をもとに詳細化し, それらの間の関係を証明する. 得られた各クラスの定理モジュールに対し, それを継承する一つの定理モジュールを生成し, 異なるクラスのオペレータの関係を証明する. これによって, その領域の理論を生成することができる.

定理モジュールの構築は, クラスモデルの生成, 関数の定義, 関数への事前事後条件の付加, HOL での証

明, 型の生成というステップを踏む.

実験として, 貸出システムについて定理モジュールを構築した. その結果, この手法においては領域全体として一つの定理モジュールを作ることは可能であるが, 領域から再利用可能なクラスを抽出することが難しいことが分かった. これはあるクラスの関数を定義する際, その関数呼出しに参加するクラスをすべて関数の引数に取る必要があるからである. また, 多くの定理はオブジェクトへのポインタが NULL でないことを前提条件としなければならず, これが型生成を困難にしていることが分かった.

謝辞

本研究を行うにあたり、片山卓也教授、青木利晃助手には熱心なご指導を頂きました。また、研究室の先輩方には様々な助言を頂きました。深く感謝致しております。

参考文献

[1] 青木利晃, 立石孝明, 片山卓也：定理証明技術のオブジェクト指向分析への応用, コンピュータソフトウェア,Vol.18,No.4(2001),pp.18-47

[2] University of Cambridge Computer Laboratory：The HOL System Description,revised edition,1991

[3] 青木利晃, 片山卓也：オブジェクト指向方法論のための形式的モデル, コンピュータソフトウェア,Vol.16,No.(1999),pp.12-32