ソフトウェア工学 138－4
（2002．7．11）

定理証明システムHOLにおけるオブジェクト指向理論の構築

矢竹健朗† 青木利晃†† 片山卓也†

高階述語論理の定理証明システム HOL は, 自然数, リスト, 集合といった厳密な数学的性質の証明に適用されてきた. HOL の高階性がもたらす高い記述能力やタクティックによる自動証明機能は, ソフトウェア検証にも有効であると考えられる. しかし, HOL には, 原始的な理論しか存在せず, これらを用いて直接アプリケーションレベルの検証を行うのは難しい. そこで本研究では, ソフトウェア検証の基礎となるオブジェクト指向の理論を HOL 上に構築した. 本稿では, そのオブジェクト指向モデルの定義と, 理論の HOL への実現法を述べる.

Constructing an Object-Oriented Theory in HOL

Kenro Yatake,† Toshiaki Aoki †† and Takuya Katayama†

HOL has been applied to proofs in mathematical theories such as natural numbers, list, set, and so on. We expect that the higher-orderness of the logic and the proof automation by tactics will also work well in the software verification. However, existing theories in HOL are too primitive to be used directly in the application level. Therefore, we constructed an Object-Oriented theory in HOL which is the groundwork for the software verification. In this paper, we present the definition of the OO model and how to implement the theory in HOL.

1. まえがき

HOL は高階述語論理の定理証明システムであり, 自然数, リスト, 集合といった厳密な数学的性質の証明に適用されてきた. HOL はこれらの数学的性質の証明だけでなくソフトウェア検証においても有効であると考えられる. 高階性は一般的な概念の記述に有効であり, また, タクティックは証明パターンに応じて柔軟に組み合わせることが可能であり, 強力な自動証明機能を実現することができる.

しかし, HOL に存在する理論は原始的なものが多く, これらを用いて, アプリケーションレベルでの証明を行うと, 非常に多くの証明ステップを要してしまう. また, 原始的な関数によって記述される命題はどのような意味を持つのか判別しにくい.

本研究では, ソフトウェア検証のための土台として, オブジェクト指向の理論を HOL 上に構築した. この理論は HOL の既存の理論から推論することによって構築しており, 無矛盾であることが保証されている.

本論文の構成は以下の通りである. 2. で HOL の概略を述べる. 3. で本研究で扱うオブジェクト指向モデルを定義し, 4. でそのモデルから実体化するシステムの理論を導入する. 5. でシステムの理論の HOL における構築法を示す. 6. でオブジェクト指向理論のための HOL ライブラリについて述べ, 7. で実験結果を考察する. 8. で関連研究と本研究を比較し, 最後に, まとめと今後の課題を述べる.

† 北陸先端科学技術大学院大学 情報科学研究科
†† 北陸先端科学技術大学院大学 情報科学研究科/科学技術振興事業団さきがけ研究２１

2. HOL

2.1 HOL における理論

HOL において理論は, 型, 定数, 公理, 定理の４つの要素から構成される. 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`pplamb` を起点とした８つの健全な推論規則により構築されている. HOL ユーザに初期に与えられる理論は`HOL` と呼ばれ, ブール代数, 自然数, リストといった基本的な理論を含んでいる. `HOL` を拡張することにより, ユーザ独自の理論を生成することが可能である. 拡張により生成される理論は親子関係をなす.

2.2 証 明 法

証明はインタプリタと対話的に行われ, Forward Proof と Goal Oriented Proof の２つの証明法がある. Forward Proof は公理とすでに証明された定理に導出規則を適用して新たな定理を導くという証明法で

ある．一方, Goal Oriented Proof は, 証明対象の命題とその前提条件をゴールに設定し, タクティックにより分解していくという証明法である．

2.3 タクティック

タクティックは Goal Oriented Proof におけるゴール分解機能である．証明の対象となっているゴール`G`を複数のサブゴール`G1`,...,`Gn` に分解する．以下, 代表的なタクティックを示す．ゴールは仮定と命題の組で, `[u]v` により表現する．

- `CONJ_TAC`
  ゴール`[u]P/\Q` を 2 つのサブゴール`[u]P`, `[u]Q` に分解する．
- `EQ_TAC`
  ゴール`[u]P=Q` を 2 つのサブゴール`[u]P==>Q`, `[u]B==>A` に分解する．
- `INDUCT_TAC`
  自然数に関する命題を, 数学的帰納法の初期段階と機能段階に分解する．つまり, ゴール`[u]!n.P` を 2 つのサブゴール`[u]P[0/n]`, `[u,P]P[SUC n'/n]` に分解する．
- `REWRITE_TAC`
  定理のリスト`[T1,...,Tn]` を引数にとり, `T1`,...,`Tn` によりゴールの書き換えを行う．

2.4 タクティカル

タクティカルは, 既存のタクティックを組み合わせてより強力なタクティックを作る機能である．証明のパターンに応じてタクティックを組み合わせることにより, そのパターンを一度に証明するタクティックを作ることができる．以下, 代表的なタクティカルを示す．

- `THEN`
  タクティックの逐次適用を実現する．`T1`,`T2` をタクティックとすると, `T1 THENL T2` は,「まず, ゴールに`T1` を適用し, それにより生成されるすべてのサブゴールに`T2` を適用する」というタクティックとなる．
- `REPEAT`
  タクティックの反復適用を実現する．`T` をタクティックとすると, `REPEAT T` は,「ゴールに`T` を適用可能な限り繰り返し適用する」というタクティックとなる．

2.5 ライブラリ

HOL では, 特定の理論の証明を支援するためのタクティックや導出規則がライブラリとして提供されている．算術理論のライブラリには, 自然数に関する定理を自動証明するためのタクティックが用意されている．独自のライブラリを作成することも可能である．

2.6 型の生成

HOL では既存の型の部分集合から新しい型を生成することができる．型生成は次の手続きにより行う．

(1) 新しい型を表現する部分集合を定める特徴述語を定義する．
(2) 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．
(3) ML 関数`new_type_definition` により新しい型を生成する．

`new_type_definition` は部分集合と新しい型との間に全単射が存在することを表明する．リスト型`'a list` は自然数とペアの型`(num->'a)#num` の部分集合から生成されている（図 1）．

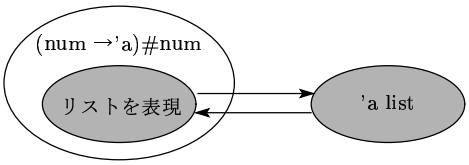

図 1 リスト型の表現

3. オブジェクトモデル

本研究では, オブジェクト指向の基本的な枠組みであるクラスの概念について理論を構築する．モデルの構成要素はクラス, 属性, 関数であり, 属性, 関数, 及びクラスから実体化するオブジェクトは型を持つ．

定義 3.1 (オブジェクトモデル)
オブジェクトモデルに導入する集合を以下に定める．

- $SystemID$：システム識別子の集合
- $ClassID$：クラス識別子の集合
- $AttrID$：属性識別子の集合
- $FunID$：関数識別子の集合
- $Type$：型の集合

オブジェクトモデルを以下に定義する．

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

ただし,

- $C \subseteq ClassID$
- $A \subseteq AttrID$
- $F \subseteq FunID$
- $T \subseteq Type$
- $\mathcal{M} = \{\mathcal{M}_{attr}, \mathcal{M}_{fun}\}$
- $\mathcal{M}_{attr} : ClassID \to Pow(AttrID)$
- $\mathcal{M}_{fun} : ClassID \to Pow(FunID)$

- $\mathcal{T} = \{\mathcal{T}_{class}, \mathcal{T}_{attr}, \mathcal{T}_{fun}\}$
- $\mathcal{T}_{class} : ClassID \to Type$
- $\mathcal{T}_{attr} : AttrID \to Type$
- $\mathcal{T}_{fun} : FunID \to Type$

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．$\mathcal{M}_{fun}$ はクラス識別子とそれが持つ関数集合を対応付ける．$\mathcal{T}_{class}$ 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．同じクラスから実体化する2つのオブジェクトは同じ型を持つとする．$\mathcal{T}_{attr}$ は属性識別子とその型を対応付ける．$\mathcal{T}_{fun}$ は関数識別子とその関数の型を対応付ける．

4. システムの理論

オブジェクトモデルから実体化するシステムについて理論を導入することにより，モデルの意味論を定義する．

システムの理論は $S = (system, init, Op, Ax)$ の4つ組として定義する．$system$ はシステムの状態を表すアリティ0の抽象データ型，$init$ はシステム初期値を表す $system$ 型の要素である．Op は $system$ 上のオペレータの集合であり，OM の構成要素に対応して導入される．Ax は Op に関する公理の集合である．

システムの理論がどのようなものであるかはスタックの理論を考えると分かりやすい．スタックの理論においては，スタックはアリティ1の抽象データ型`stack`として表現され，空スタック`EMP`をその要素として持つ．また，`stack`型上のオペレータとして`PUSH`，`POP`があり，`!x s. POP (PUSH x s) = s`といった公理が存在する．システムの理論もスタックと同様な理論である．

4.1 オペレータの導入

オブジェクトモデル OM の構成要素に対応して，Op の要素を定義する．オペレータには，存在検査述語，オブジェクト生成関数，オブジェクト数取得関数，属性取得関数，属性更新関数の5種類がある．

- 存在検査述語
  $c_i \in C$ について，存在検査述語
  $class_i\_exists : \mathcal{T}_{class}(c_i) \to system \to bool$
  を Op の要素とする．$class_i$ は c_i の名前とする．
  $class_i\_exists\ o\ s$ はクラス c_i のオブジェクト o がシステム s 内に存在するかどうかを返す．
- オブジェクト生成関数
  $c_i \in C$ について，オブジェクト生成関数
  $class_i\_gen : system \to \mathcal{T}_{class}(c_i) * system$
  を Op の要素とする．$class_i$ は c_i の名前とする．
  $class_i\_gen\ s$ はシステム s にクラス c_i に属す新たなオブジェクトを生成し，そのオブジェクトと生成後のシステムのペアを返す．
- オブジェクト数取得関数
  $c_i \in C$ について，オブジェクト数取得関数
  $class_i\_num : system \to num$
  を Op に導入する．$class_i$ は c_i の名前とする．
  $class_i\_num\ s$ はシステム s 内に存在するクラス c_i のオブジェクト数を返す．
- 属性取得関数
  クラス c_i が属性 a_j を持つとき，属性取得関数
  $class_i\_get\_attr_j :$
  $\mathcal{T}_{class}(c) \to system \to \mathcal{T}_{attr}(a_j)$
  を Op の要素とする．$class_i$ は c_i の名前，$attr_j$ は a_j の名前とする．
  $class_i\_get\_attr_j\ o\ s$ はクラス c_i に属すオブジェクト o の属性 a_j の値を返す．
- 属性更新関数
  クラス c_i が属性 a_j を持つとき，属性更新関数
  $class_i\_fun\_attr_j :$
  $(\mathcal{T}_{attr}(a_j) \to \mathcal{T}_{attr}(a_j)) \to$
  $\mathcal{T}_{class}(c) \to system \to system$
  を Op の要素とする．$class_i$ は c_i の名前，$attr_j$ は a_j の名前とする．
  $class_i\_fun\_attr_j\ f\ o\ s$ はクラス c_i に属すオブジェクト o の属性 a_j に関数 f を適用し，適用後のシステムを返す．

4.2 公理の導入

Op の要素について，以下の公理を Ax に導入する．

[A-1] $\forall f\ o\ s.\ class_i\_exists\ o\ s \Rightarrow$
$(class_i\_get\_attr_k\ o\ (class_i\_fun\_attr_k\ f\ o\ s)$
$= f\ (class_i\_get\_attr_k\ o))$

[A-2] $\forall f\ o_1\ o_2\ s.\ \neg(o_1 = o_2) \Rightarrow$
$(class_i\_get\_attr_k\ o_1\ (class_i\_fun\_attr_k\ f\ o_2\ s)$
$= class_i\_get\_attr_k\ o_1)$

[A-3] $\forall f\ o_1\ o_2\ s.$
$class_i\_get\_attr_k\ o_1\ (class_i\_fun\_attr_l\ f\ o_2\ s)$
$= class_i\_get\_attr_k\ o_1\ (k \neq l)$

[A-4] $\forall f\ o_1\ o_2\ s.$
$class_i\_get\_attr_k\ o_1\ (class_j\_fun\_attr_l\ f\ o_2\ s)$
$= class_i\_get\_attr_k\ o_1\ (i \neq j)$

[A-5] $\forall f\ o1\ o2\ s.$
$class_i\_exists\ o_1\ (class_j\_fun\_attr_k\ f\ o_2\ s) =$
$class_i\_exists\ o_1\ s$

[A-6] $\forall f\ o\ s.\ \neg(class_i\_exists\ o\ s) \Rightarrow$

$(class_i\_fun\_attr_k\ f\ o\ s = s)$

[A-7] $\forall o\ s.\ class_i\_exists\ o\ s \Rightarrow (class_i\_get\_attr_k\ o\ (snd\ (class_i\_gen\ s)) = class_i\_get\_attr_k\ o\ s)$

[A-8] $\forall o\ s.\ class_i\_get\_attr_k\ o\ (snd\ (class_j\_gen\ s)) = class_i\_get\_attr_k\ o\ s\ (i \neq j)$

[A-9] $\forall s.\ (p = class_i\_gen\ s) \Rightarrow class_i\_exists\ (fst\ p)\ (snd\ p) \wedge \neg(class_i\_exists\ (fst\ p)\ s)$

[A-10] $\forall o\ s.\ class_i\_exists\ o\ s \Rightarrow class_i\_exists\ o\ (snd\ (class_i\_gen\ s))$

[A-11] $\forall o\ s.\ class_i\_exists\ o\ (snd\ (class_j\_gen\ s)) = class_i\_exists\ o\ s\ (i \neq j)$

[A-12] $class_i\_num\ init = 0$

[A-13] $\forall s.\ class_i\_num\ (snd\ (class_i\_gen\ s)) = class_i\_num\ s + 1$

[A-14] $\forall s.\ class_i\_num\ (snd\ (class_j\_gen\ s)) = class_i\_num\ s\ (i \neq j)$

[A-15] $\forall f\ o\ s.\ class_i\_num\ (class_j\_fun\_attr_k\ f\ o\ s) = class_i\_num\ s$

各公理の意味を以下に示す.

[A-1] クラス c_i に属すオブジェクト o の属性 a_k に関数 f を適用後, 同じ属性を取得したとき, その値は, 適用前に取得する値に f を適用した値である. ただし, o がシステム内に存在することが条件である.

[A-2] クラス c_i に属すオブジェクト o_2 の属性 a_k に関数 f を適用後, o_1 とは異なるオブジェクト o_2 が属性 a_k を取得するとき, その値は適用前に取得する値と等しい.

[A-3] クラス c_i に属すオブジェクト o_1 の属性 a_k に関数 f を適用後, 同じクラスに属すオブジェクト o_2 の, a_k とは異なる属性 a_l を取得するとき, その値は適用前に取得する値と等しい.

[A-4] クラス c_i のオブジェクトの属性に関数を適用しても, c_i とは異なるクラス c_j のオブジェクトの属性は不変である.

[A-5] システムに存在するオブジェクトは, あるオブジェクトに対して関数が適用された後も存在する.

[A-6] システム内に存在しないオブジェクトに対する関数適用は無効である.

[A-7] オブジェクト o がシステムに存在するならば, 同じクラスのオブジェクトが新たに生成された直後に取得する o の属性の値は, オブジェクト生成前に取得する値と等しい.

[A-8] オブジェクト o が, それとは異なるクラスのオブジェクトが生成された直後に取得する属性の値は, オブジェクト生成前に取得する値と等しい.

[A-9] あるオブジェクトが生成されたとき, そのオブジェクトは生成後のシステムに存在する. また, 生成前のシステムには存在しない. つまり, 生成されるオブジェクトは生成前に存在するオブジェクトとは異なるオブジェクトである.

[A-10] オブジェクトがシステムに存在するならば, 同じクラスのオブジェクトが生成された直後も存在する.

[A-11] あるクラスのオブジェクトが生成されても, 他のクラスのオブジェクトが存在するという事実は不変.

[A-12] システムの初期値においてはどのクラスのオブジェクトも存在しない.

[A-13] あるクラスのオブジェクトが生成されたとき, そのクラスのオブジェクト数は 1 増加する.

[A-14] あるクラスのオブジェクトが生成されても, 他のクラスのオブジェクト数は不変.

[A-15] オブジェクトへの関数適用はどのクラスのオブジェクト数にも影響しない.

4.3 クラス関数の定義

オブジェクトモデルにおいてはクラスは関数を持つ. これらの関数の振る舞いは, 開発者が定義しなければならない. 関数はモデルを決定することによって与えられる属性取得関数, 属性更新関数, オブジェクト生成関数を用いて定義する.

いま, モデルにおいて, カウンタクラス`counter` が属性としてカウント数`number:num`, 関数としてカウント数をリセットする関数`reset:unit->unit` とカウント数を 1 増加させる関数`inc:unit->unit` を持つとする. HOL において`reset`, `inc` は, `number` に対応して導入される属性更新関数`counter_fun_number` を用いて次のように定義することができる.

```
counter_reset : counter -> system
|- !i s. counter_reset i s =
   counter_fun_number (\x.0) i s
counter_inc : counter -> system
|- !i s. counter_inc i s =
   counter_fun_number SUC i s
```

また, カウント数が 8 以下のときカウントし, 9 以上のときリセットする関数`loop_inc:unit->unit` は, 属性取得関数`counter_get_number` を用いて次のように定義することができる.

```
counter_loop_inc : counter -> system
```

```
|- !i s. counter_loop_inc i s =
   ((counter_get_number i s <= 8) =>
    counter_inc i s | counter_reset i s)
```

このように開発者は, 属性取得関数, 属性更新関数, オブジェクト生成関数を組み合わせてクラスの関数の振る舞いを定義していく.

4.4 オブジェクト間通信

クラスの関数から他のクラスの関数を呼び出すこともできる. これは, クラスが属性としてオブジェクトを保持することによって実現できる. つまり, 関数定義の中で属性として保持しているオブジェクトを取得し, そのオブジェクトの関数を呼び出せばよい.

いま, 図書館クラスが属性として利用者オブジェクトのリスト`users:user list`を保持しているとする. このとき, 全利用者の貸出数の総和をとる関数を次のように定義することができる.

```
lib_get_user_loan_sum : lib -> system -> num
|- !i s. lib_get_user_loan_sum i m s =
   let l = lib_get_users i s in
   SUM (MAP (\j.user_get_loan_num j s) l)
```

`lib_get_users`は図書館クラスの属性`users`に対して与えられる属性取得関数であり, `user_get_loan_num`は利用者クラスが持つ, 貸出数を取得する関数である. `MAP`により, `l`に含まれる利用者オブジェクトすべてについて貸出数を取得し, `SUM`でその総和をとる.

5. HOL への実装

本研究では, システムの理論を HOL にすでに存在する理論から推論することにより構築する. HOL では, 8つの推論規則により導出される理論は無矛盾となる.

HOL 上にオブジェクトシステムの理論を構築する方針は, HOL 上にオブジェクト指向プログラムの実行環境を定義し, その実行環境について事実を導くことである. Java では, オブジェクトが生成されるごとに, そのデータはヒープ領域に格納されていく. 本研究では, このメモリ構造の基本的な原理をリストによって定義した.

5.1 メモリ理論の構築

メモリ理論の構築は以下の流れで行う.

(1) リスト理論によるオペレータの定義
(2) オペレータに関する定理の証明
(3) メモリ型の生成

5.1.1 オペレータの定義

単純なメモリ構造をリストにより構築する. メモリに格納されるデータの型を`'a`とし, メモリ空間全体をデータの配列`'a list`として表現する. データのアドレスはリストのインデックスによって表現する (図2). メモリに対する操作として, アドレスの有効性

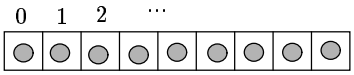

図 2 リストによるメモリ構造の表現

検査, データの取得, 更新, 追加の4つを導入する. それぞれ, リスト上の関数`EXISTS`, `GET`, `SET`, `ADD`として次のように定義する.

```
EXISTS : num -> 'a list -> bool
|- !i l. EXISTS i l = i < LENGTH l
GET : num -> 'a list -> 'a
|- (!l. (GET 0 l = HD l)) /\
   (!n l. GET (SUC n) l = GET n (TL l))
SET : num -> 'a -> 'a list -> 'a list
|- (!x l. (SET1 0 x l = CONS x (TL l))) /\
   (!n x l. SET1 (SUC n) x l =
    CONS (HD l) (SET1 n x (TL l)))
|- !i x l. SET i x l =
   (EXISTS i l => SET1 i x l | l)
ADD : 'a -> 'a list -> num # 'a list
|- !x l. ADD x l = (LENGTH l,SNOC x l)
```

`EXISTS`は, メモリ`l`のアドレス`i`にデータが存在するかを検査する. 指定されたアドレスにデータが存在するということは, アドレスがリスト長より小さいということである. `GET`は, メモリ`l`のアドレス`i`に格納されているデータを取得する. `SET`は, メモリ`l`のアドレス`i`のデータを`x`に更新する. 指定されるアドレスが無効であればメモリに変更を加えない. `ADD`は, メモリ`l`に新たなデータ`x`を追加し, `x`を格納したアドレスと, 追加後のメモリを出力する. 新しいデータはリストの最後尾に追加する. これは`CONS`と対象の動作をする`SNOC`により行う.

5.1.2 定理の証明

定義したオペレータについて定理を証明する.

`EXISTS`, `GET`, `SET`の間には次の定理が成り立つ.

```
|- !i x l. EXISTS i l ==>
   (GET i (SET i x l) = x)
```

これは, 「アドレス`i`にデータが存在するならば, アドレス`i`のデータを`x`に設定した直後, アドレス`i`のデータを取得するとその値は`x`」を意味する. この定理は, `i`について数学的帰納法を適用し, 生成されるサブゴールに対し, `l`についてリスト構造に関する帰納法を適用することにより証明することができる.

GET とSET の間には次の定理が成り立つ.

```
|- !x i j l. ~(i = j) ==>
   (GET i (SET j x l) = GET i l)
```

これは,「アドレスj のデータをx に更新した直後, j とは異なるアドレスi のデータを取得するとその値は更新前に取得する値と等しい」を意味する. つまり, 個々のアドレスのデータは独立しており, あるアドレスのデータの変更は他のアドレスのデータに影響を与えない.

EXISTS, SET の間には次の 2 つの定理が成り立つ.

```
|- !x i j l.
   EXISTS i (SET j x l) = EXISTS i l
|- !x i l. ~(EXISTS i l) ==> (SET i x l = l)
```

一つ目は,「アドレスj のデータをx に更新後にアドレスi のデータが存在することは, 更新前にi のデータが存在することと同値」を意味する. つまり, データの更新によりデータがメモリから消去されることはない. 二つ目は,「アドレスi にデータが存在しないならば, i のデータを更新しようとしても無効となる」を意味する.

EXISTS とADD の間には次の二つの定理が成り立つ.

```
|- !x l. let p = ADD x l in
   EXISTS (FST p) (SND p) /\
   ~(EXISTS (FST p) l)
|- !x i l. EXISTS i l ==>
   EXISTS i (SND (ADD x l))
```

一つ目は,「新たにデータが追加されたとき, そのアドレス (FST p) にはデータが存在する. かつ, 追加前はそのアドレスにはデータが存在しない」を意味する. この定理から, 追加されるデータは新しいメモリ領域に格納されることがわかる. 二つ目は,「アドレスi にデータが存在するならば, 新たなデータが追加された直後もアドレスi にデータが存在する」を意味する.

次の 3 つの定理はメモリのサイズに関する定理である. メモリのサイズはリストの長さによって表現する.

```
|- LENGTH [] = 0
|- !x l. LENGTH (SND (ADD x l)) =
   LENGTH l + 1
|- !i x l. LENGTH (SET i x l) = LENGTH l
```

それぞれ,「空メモリのサイズは 0」,「新たなデータが追加されたとき, メモリサイズは 1 増加する」,「データの更新により, メモリサイズは変化しない」を意味する.

5.1.3 メモリ型の生成

リスト理論において定義したオペレータはメモリを表現する部分集合を決定する. この部分集合を表す特徴述語を定義し, メモリ型mem を生成した (図 3).

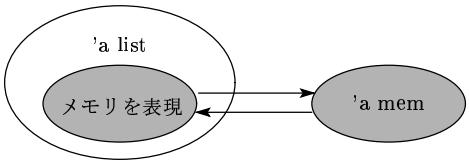

図 3 メモリ型の表現

以下はmem 型上の定数, オペレータであり, それぞれリスト理論における[], EXISTS, GET, SET, ADD, LENGTH に対応する.

```
MEM_EMP     : 'a mem
DATA_EXISTS : num -> 'a mem -> bool
GET_DATA    : num -> 'a mem -> 'a
SET_DATA    : num -> 'a -> 'a mem -> 'a mem
ADD_DATA    : 'a -> 'a mem -> num # 'a mem
DATA_SIZE   : 'a mem -> num
```

これらのオペレータについて以下の公理が成り立つ.

```
|- !i x m. DATA_EXISTS i m ==>
   (GET_DATA i (SET_DATA i x m) = x)
|- !x i j m. ~(i = j) ==>
   (GET_DATA i (SET_DATA j x m) =
    GET_DATA i m)
|- !x i j m. DATA_EXISTS i (SET_DATA j x m)
   = DATA_EXISTS i m
|- !x i m. ~(DATA_EXISTS i m) ==>
   (SET_DATA i x m = m)
|- !x i m. DATA_EXISTS i m ==>
   (GET_DATA i (SND (ADD_DATA x m)) =
    GET_DATA i m)
|- !x m. let p = ADD_DATA x m in
   DATA_EXISTS (FST p) (SND p) /\
   ~(DATA_EXISTS (FST p) m)
|- !x i m. DATA_EXISTS i m ==>
   DATA_EXISTS i (SND (ADD_DATA x m))
|- !i x m.
   MEM_SIZE (SET_DATA i x m) = MEM_SIZE m
|- MEM_SIZE (MEM_EMP:'a mem) = 0
|- !x m. MEM_SIZE (SND (ADD_DATA x m)) =
   MEM_SIZE m + 1
```

5.2 システム理論の構築

オブジェクトのデータは, その複数の属性をタプルでまとめたものとして表現する. 例えば, 本クラスが属性としてnum 型の「登録番号」とstring 型の「タ

イトル」と「著者名」を持つとき, 本オブジェクトのデータは`num#string#string` となる.

システムには複数のクラスのオブジェクトが存在する. それぞれのクラスに一つづつメモリを割り当て, それらのクラスから実体化するオブジェクトはその割り当てられたメモリに格納する. 本クラスに割り当てられるメモリの型は`(num#string#string)mem` ということになる. システムの型`system` は, 各クラスのメモリをタプルでつないだものとして表現する. つまり, n 個のクラスのメモリの型が`t1`,...,`tn` とすると, システム型は`t1#...#tn` で表現される（図 4）.

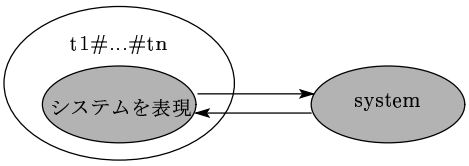

図 4 システム型の表現

タプルに含まれる要素の型はそれぞれ異なるので, 一般的なシステムの理論を構築することはできない. しかし, メモリ理論から具体的なシステムの理論を構築する過程はパターン化しているので, 本研究ではそれを自動化した.

6. ライブラリ obj

オブジェクト指向理論における証明を支援するための HOL ライブラリを構築した. このライブラリは主に次の機能を持つ.

- 理論の自動生成
  クラス, 属性, 型等のモデルの情報が記述されたファイルを読み込み, その理論を HOL 上に構築するための HOL ファイルを自動生成する. これは ML 関数`Obj.mk_theory:string->unit` が行う.
- 公理の自動証明
  システムの理論に含まれる公理数は非常に多く, クラス数, 属性数の二乗に比例して増加する. 本研究では, 莫大な数の公理をすべて自動生成することはせず, 代わりに [A-1] から [A-15] のそれぞれの公理を証明するための 15 種類の ML 関数を用意した. 次に示すのは, 公理 [A-1] を証明する関数である.

  ```
  Obj.prove_a1:
  {Ex:term,Get:term,Fun:term} -> thm
  ```

  この関数は入力として存在検査述語`Ex`, 属性取得関数`Get`, 属性更新関数`Fun` をとり, それらの間に公理 [A-1] が成り立つならばその公理を証明して出力する.

  ```
  > Obj.prove_a1{Ex=``user_exists``,
    Get=``user_get_uid``,Fun=``user_fun_uid``};
  - val it = !f i s. user_exists i s ==>
    (user_get_uid i (user_fun_uid f i s) =
    f (user_get_uid i s)) : thm
  ```

- タクティック
  Goal oriented proof により証明を行う際, 公理生成関数を用いて, そのつど必要な公理を取得するのは手間がかかる. これを解決するために, ゴールの命題の形に応じて, 必要な公理を生成し, 命題の書き換えを自動的に行うタクティックを用意した.

7. 実験と考察

本研究で構築したオブジェクト指向理論を用いて図書館システムの検証を行った. 対象とした図書館システムは C++で 400 行程度のシステムであり, 本の貸出, 返却, 本・利用者の登録, 削除といった基本的な機能に加え, 本の予約, 予約キャンセル, 予約者に対する本割り当てといった比較的高度な機能も持つ. C++のコードを HOL の関数として定義しなおし, 「貸出手続き前後で, 利用者の貸出総数と, 本の貸出総数は等しい」という定理を証明した. 実験を通して直面した問題点を以下に示す.

- 式の煩雑さ
  Goal Oriented Proof により証明を行う際, ゴールの命題は分解するにつれて大きな式となり, 煩雑化する. これは, 公理を用いて書き換えるためには, 命題を属性操作関数, 属性更新関数, オブジェクト生成関数が現れるまで分解しなければならないことが原因である. これを解決するためには, 命題を複数の細かな補題に分けて証明する必要がある. 「貸出手続き前後で利用者の貸出総数と本の貸出総数は等しい」という定理を証明する際も, 「貸出手続き前後で貸出を行った利用者の貸出数が 1 増加する」, 「貸出手続き前後で貸出を行わなかった利用者の貸出数は等しい」, といった補題を証明している. 実際はさらに細かく, 個々のクラス関数ごとに補題を証明している. このように補題に分けて証明することは, 式の煩雑化を防ぐとともに, 補題の再利用という点でも有効である.
- 不変的性質の証明
  本研究で構築した理論においては, 「貸出前後で利

用者の貸出総数と本の貸出総数は等しい」といった, 関数適用前後でのシステムの状態変化に関する定理しか証明することができない. 実際は「利用者の貸出総数と本の貸出総数は等しい」という性質は, 図書館についてどのような関数が適用されても常に成り立っている性質である.

このような, システムについての不変的な性質を証明するためには, システムの状態遷移に関して帰納的に証明を行う必要がある. つまり, システム`s` の状態を変化させる関数`f1`,...,`fn` が存在するとき,

```
!P s. (P init /\
       !s. P s ==> P(f1 s) /\
          ...
       !s. P s ==> P(fn s)) ==> P s
```

という述語を前提条件とする必要がある. しかし, 一般にこの前提条件からは矛盾が導かれるので, 公理系の信頼性が弱まってしまう. また, 帰納段階は n 個存在するので, n が大きくなると証明コストが大きくなってしまうという問題もある.

8. 関連研究

[1] は, 拡張レコードという概念を用いてオブジェクト指向の理論を HOL に実現している. 拡張レコードは`{x=a,y=b,...}`のように表されるデータであり, 「`...`」の部分に新たな要素を追加し, `{x=a,y=b,z=c,...}`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 `num#num#bool#'a` で表現されている. `'a` を`num#'a` で詳細化することにより, 拡張を行っている.

クラスは属性`x:num`, `y:num` を持つとき, 拡張レコード`{x:num,y:num,...}`として表現される. オブジェクトは`{x=1,y=2}`のように, クラスの拡張レコードを具体的なレコードに拡張したものとして表現される. 関数はクラスの拡張レコードに対して定義され, `move {x,y} dx dy = {x+dx,y+dy}`といった属性の更新が可能である. 継承は, クラスの拡張レコードに要素を追加することにより実現している. 関数のオーバーライドは, スーパークラスの関数をサブクラスの関数が補助関数として用いることで実現している. 関数定義の仕方を工夫することで動的束縛の原理を実現し, スーパークラスの定理をサブクラスが補題として用いることができるようになっている（証明の継承）.

本研究と [1] の大きな違いは, オブジェクト指向の「横」の概念を扱うか, 「縦」の概念を扱うかである. [1] ではオブジェクトは拡張レコード単体として表現されているため, 複数のオブジェクトが協調動作するという概念がない. 本研究では, オブジェクト集合をシステムという一つの枠組みの中で扱っており, オブジェクトの協調動作を実現することができる. したがって, オブジェクト間の関係や, オブジェクトの存在, 個数に関する検証が可能である.

[1] では, クラスの継承の概念があり, クラス階層の上位の証明を下位で再利用するという原理が実現されている. 本研究では, 複数のクラスをまとめて一つのシステムとして扱っているので, クラスが単位となるような継承などの概念を扱うのは難しい.

9. まとめと今後の課題

本研究では, オブジェクト指向理論を HOL 上に実現した. オブジェクト指向モデルは, クラス, 属性, 関数から構成される. モデルの意味論は, モデルから実体化するシステムについて理論を導入することにより与えた. システムの理論は HOL の既存の理論から推論して構築した. システムの理論はメモリの基本原理に基づいている. また, オブジェクト指向理論における証明を支援するための HOL ライブラリを作成した. このライブラリを使って, システムの理論の自動生成が可能である.

今後は, 継承や集約といったオブジェクト指向の基本概念を理論に導入していく. また, オブジェクト消滅関数やオブジェクトアクセス数カウンタをメモリ理論に導入し, システムの理論の拡張を行う.

参考文献

1) Wolfgang Naraschewski, Markus Wenzel: Object-Oriented Verification based on Record Subtyping in Higher-Order Logic, 1998
2) 青木利晃, 立石孝彰, 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
3) 青木利晃, 立石孝彰, 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.1(1999), pp.12-32
4) University of Cambridge Computer Laboratory:The HOL System Description, revised edition, 1991
5) D'souza, Wills : Objects, Components, and Frameworks with UML : The Catalysis Approach