オブジェクト列に対するループ文の推論規則

矢竹 健朗　片山 卓也

ホーア論理に基づくプログラム検証では，while 文の推論規則を適用する際に，ユーザがループ不変表明を与えなければならない．ループ不変表明の発見は一般に難解であり，これがプログラム検証が実用化される上での大きな障壁となっている．本稿では逐次的オブジェクト指向プログラムについて，オブジェクト列に対するループ文が，特定の条件下では不変表明によらない特別な推論規則で還元できることを示す．これにより，ユーザが不変表明を与える手間が減り，プログラム検証の効率を改善することができる．

1 はじめに

ホーア論理 [2] はプログラムの正当性を証明するための形式体系であり，古典的ではあるが確立した体系となっている．実際，現在の多くのプログラム検証法はその考え方に基づいている．ホーア論理では，プログラム P に対し，その事前事後表明を表す論理式 A, B を $\{A\}P\{B\}$ のように与え，代入文の公理や if 文，while 文の推論規則を用いて，プログラムの正しさを表明の正しさに還元していく．最終的に得られる表明の集合は検証条件と呼ばれ，これを証明すればもとのプログラムが正しいことが証明される．検証条件生成器とはこの一連の過程を自動化支援するソフトウェアである．この手法の難しさは，while 文の推論規則を用いる際に，ループ不変表明と呼ばれる中間的な表明を与えなければならないという点にある．適切なループ不変表明を自動的に抽出することは難しく，通常，ユーザが与えなければならない．この負担がプログラム検証が実用化される上での大きな障壁となっている．しかし，プログラムの正当性を厳密に検証しようとする場合，根本的にはホーア論理の考え方に行き着くため，その効率を改善することは重要である．

本稿では逐次的オブジェクト指向プログラムについて，オブジェクト列に対するループ文が，特定の条件下では不変表明によらない推論規則で還元できることを示す．本稿で着目するループ文は，オブジェクト列に対し，その列に含まれる要素に対し順番にメソッドを呼び出すといった構造を持つ文である．オブジェクト列はオブジェクト指向プログラムにおいて頻繁に使用されるデータ構造であり，これを機械的に扱えるようにすることは，プログラム検証の効率を改善する上で重要である．

本稿では，本質を分かりやすくするために，Java や C++などの特定の言語ではなく，抽象的なオブジェクト指向言語を想定し，それに対して推論規則を定義する．推論規則の健全性は，高階述語論理の定理証明器 HOL [1] により証明する．

本稿の構成は次の通りである．2 節において，本稿で導入する推論規則の原理を述べる．3 節において，オブジェクト指向言語と表明言語を導入する．4 節において，推論規則を導入し，その健全性を示す．5 節において，例題の証明を行う．6 節において，考察を述べる．7 節において，まとめと今後の課題を述べる．

Kenro Yatake，北陸先端科学技術大学院大学，Japan Advanst Institute of Science and Technology
Takuya Katayama，北陸先端科学技術大学院大学，Japan Advanst Institute of Science and Technology

2 概要

本稿で導入する推論規則の原理を概説する．例として，ネットワーク接続のタイムアウトを管理するシステムを考える．このシステムは，定期的にすべての接続のタイムアウト時間を一括して減算する方式をとっているとする．つまり，図に示すように，x1,...xn が接続オブジェクトであり，それらすべてに対し順番に decTimer() というメソッドを呼び出すことにより，タイムアウト時間を 1 づつ減算する．通常この操作は，オブジェクト列に対する while ループにより実現される．

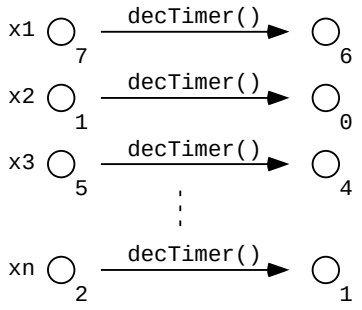


図 1 接続オブジェクト列に対するタイムアウト減算

ここで，ある 1 つの接続オブジェクトに注目し，そのオブジェクトの while ループ前後でのタイムアウト時間の変化を調べたいとする．通常は適切なループ不変表明を考え，while 文の推論規則を適用することになる．しかし，この例の場合，while 文全体を見なくても，そのオブジェクトに対して適用された単一の decTimer() メソッド呼び出しを調べるだけで，変化量が-1 であることがわかってしまう．つまり，while ループ全体で呼ばれたそれ以外の decTimer() メソッドは，そのオブジェクトに全く影響を与えていない．

以上の考察を一般化し，HOL の定理として証明すると次のようになる．

```
|- !(x:'a) (S:'b) (L:'a list)
    (P:'a->'b->bool) (G:'a->'b->'b).
   UNIQ L /\
   DIFF_OBJ_UPD_UPD G /\
   DIFF_OBJ_REF_UPD P G ==>
     (P x (FOLDL (\s x. G z s) S L) =
      if MEM x L then P x (G x S)
      else P x S)
```

x はオブジェクト参照，S は環境，L はオブジェクト列である．オブジェクト列は，可変長リストにより定義している．P はオブジェクトに対する表明であり，オブジェクトと環境を引数とし，真偽値を返す．G はオブジェクト列に対して順に適用されるメソッドであり，オブジェクトと環境を引数とし，新しい環境を返す．

結論部の等式について，

```
FOLDL (\s x. G z s) S L
```

は，while ループの実行後の環境を表す．つまり，環境 S のもとで，L に含まれる要素に対し順番にメソッド G を適用した後の環境である．したがって，等式の左辺は，ループ後のあるオブジェクト x に関する表明 P の真偽を表す．この値は，x が L の要素ならば，環境 G x S，つまり，最初の環境 S のもとで x に G を適用して得られる環境における P の値に等しい．つまり，ループ全体で繰り返し適用されたメソッド G のうち，x 以外に適用されたものは除外することができる．x が L に含まれていない場合は，S における P の値に等しい．つまり，ループの実行自体が x に影響を与えない．

この等式の成立には 3 つの条件が前提となる．まず，UNIQ L は，オブジェクト列の単一性を意味する．UNIQ は次のように定義される述語である．

```
|- (UNIQ [] = T) /\
   (UNIQ (x::L) = ~MEM x L /\ UNIQ L)
```

つまり，L には同一の要素が重複して含まれていてはならない．同一の要素が複数含まれる場合，そのオブジェクトに対してメソッドが複数回適用されるため，結局ループ不変表明を求めることが必要になる．要素が単一であれば，1 回のメソッド適用だけを調べるだけでよい．

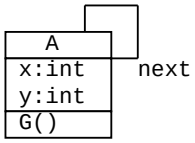
次に，DIFF_OBJ_REF_UPD は，表明 P とメソッド G に関する制約であり，次のように定義される．

```
|- DIFF_OBJ_REF_UPD P G =
   !x y S. ~(x = y) ==>
```

```
(P x (G y S) = P x S)
```

つまり，あるオブジェクト x に関する表明 P は，それとは異なるオブジェクト y に対するメソッド G の適用に影響を受けてはならない．言い換えれば，x に関する表明 P が観測するメモリ（環境）の領域は，y に対するメソッド適用により更新されてはならない．例えば図 2 のように，環状にリンクするクラス A のオブジェクト列 a1,...,a5 があるとする．この列に対し，順番にメソッド G を適用する場合，オブジェクト a3 のフィールド x, y に関する表明，例えば x=y は，1 つ前のオブジェクト a2 に対する G の適用の影響を受ける．これは，a3 のフィールド y がその前のオブジェクト a2 のフィールド x の値に書き換えられるためである．このような場合，あるオブジェクトに関する表明の真偽を問う際に，それ以外のオブジェクトに対するメソッドの適用を除外して考えることはできない．したがって，表明の観測領域が他のオブジェクトのメソッド適用に影響を受けないという条件が必要となる．

```
P : x = y
L : {a1,a2,a3,a4,a5}
```




```
void G() {
  next.y = this.x;
}
```

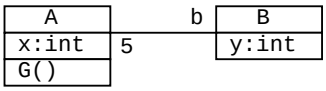
図 2 表明の観測領域が影響を受ける例

最後に，DIFF_OBJ_UPD_UPD は，メソッド G の適用順序に関する制約であり，次のように定義される．

```
|- !G. DIFF_OBJ_UPD_UPD G =
   !x y S. ~(x = y) ==>
     (G x (G y S) = G y (G x S))
```

つまり，G が異なるオブジェクト x, y に対して適用されたとき，それらの適用順序は入れ替え可能でなければならない．前の制約 DIFF_OBJ_REF_UPD は，表明が直接他のメソッドに影響を受ける場合を排除するものであるが，この制約はメソッド適用順序による間接的な影響を排除するものである．例えば図 3 について，クラス A のオブジェクト列 a1,...,a5 に対し，順番にメソッド G を適用する場合を考える．ループ後，a5 のフィールド x の値は 5 となるが，仮に，a5 に対して単独に G を適用した場合，その値は 1 となる．したがって，a5 に関する表明，例えば x>3 は，それ以前に実行された a1,...,a4 に対するメソッド適用に間接的に影響を受けることとなる．他のオブジェクトに対するメソッド適用を除外して，a5 に対するメソッド適用単独で考えることは，本来 5 番目である a5 に対するメソッド適用を最初に持ってくることであり，これを可能とするためには，表明がメソッドの適用順序に依存しないこと，つまり，メソッド適用順序が入れ替え可能であるという条件が必要となる．

```
P : x > 3
L : {a1,a2,a3,a4,a5}
```




```
void G() {
  b.y++;
  this.x = b.y;
}
```

図 3 表明が間接的に影響を受ける例

2 番目の条件と 3 番目の条件は似ているが，互いに包含関係にない条件である．実際，図 2 は，前者は満たさないが後者は満たす例であり，図 3 は，前者を満たすが後者は満たさない例である．

以上，3 つの条件下では，「while ループで順次適用されるメソッドのうち，関心のあるオブジェクトに対するメソッド適用だけを残す」といった推論が可能になる．

3 オブジェクト指向言語と表明言語

前節で述べた推論規則を言語レベルで定義するために，オブジェクト指向言語と表明言語を導入する．ここでは，形式的な文法は省略し，例を用いて説明する．

図 4 は，前節の最初に述べた接続を管理するシステムである．基本的な構文は Java と同様であるが，

変数の型は num（自然数）や bool（真偽値）などの HOL の型を使用している．これは意味論との親和性を考慮しているためである．また，オブジェクト列操作に関しては，操作の意味を明確にするために，app や filter などの高級な構文を導入している．

クラス contable, connection はそれぞれ接続表，接続を表すクラスである．connection クラスのフィールド timer はタイムアウト残り時間を表す．メソッド decTimer() は，タイムアウト残り時間を減算するメソッドである．メソッド isAlive() は，タイムアウトしていないときに真を返すメソッドである．contable クラスのフィールド connections は，管理する接続オブジェクトの列である．メソッド decTimer() は，管理するすべての接続のタイムアウト残り時間を減算し，タイムアウトした接続を削除するメソッドである．このメソッドにおいて，

```
L.app(decTimer());
```

は，connection オブジェクト列 L のそれぞれの要素に対し，順に decTimer() メソッドを適用するという意味である．また，

```
L = L.filter(isAlive());
```

は，L に含まれる要素のうち，メソッド isAlive() を適用した結果が真となるものだけを取り出し，新しい L に格納するという意味である．

以下は，この 2 行に事前事後表明を与えたものである．

```
{ L.mem(x) && (x.timer>1) && (x!=null) }
L.app(decTimer());
L = L.filter(isAlive());
{ L.mem(x) }
```

つまり，2 行の実行前に，接続表の要素で残り時間が 1 より大きい接続は，実行後も接続表に存在する．L.mem(x) は，オブジェクト x がオブジェクト列 L の要素であるという意味の式である．

4 推論規則と健全性

前節のオブジェクト指向言語と表明言語について，app の推論規則を図 5 のように導入する．結論部は，事前条件 P のもとで $L.app(G())$ を実行後，オブジェクト x に関する表明 $Q(x)$ が成り立つことを意味する．2 つの前提条件は，それぞれ，x が L の要素であるとき，ないときの条件である．前者は，x に対するメソッド $G()$ の適用後に結論 $Q(x)$ が成り立てばよい．後者は，実行を介せずに $Q(x)$ が導出できなければならない．ここで，L, Q, G は，2 節で述べた 3 つの条件を満たすものとする．言語レベルでそれらの条件を保証する方法については考察で述べる．

```
class connection {
  num timer;
  void decTimer() {
    this.timer = this.timer - 1;
  }
  bool isAlive() {
    return this.timer > 0;
  }
}

class contable {
  (connection list) connections;
  void decTimer() {
    (connection links) L;
    L = this.connections;
    L.app(decTimer());
    L = L.filter(isAlive());
    this.connections = L;
  }
```

図 4 サンプルプログラム

図 6 は，表明 $L.forall(z|Q(z))$ と $L.exists(z|Q(z))$ に関して同様の考え方で導入される推論規則である．前者は，L のすべての要素について Q が真となること，後者は，L のある要素について Q が真となることを意味する表明である．z は論理式の仮引数である．

本稿では簡単のため，G が引数を持たない場合の推論規則を示しているが，L の要素ごとに引数を与えて適用する場合についても容易に定義することが可能である．

推論規則の健全性は，HOL において次の定理とし

$$\frac{\{L.mem(x) \wedge P\}\ x.G()\ \{Q(x)\} \qquad \neg L.mem(x) \wedge P \Rightarrow Q(x)}{\{P\}\ L.app(G())\ \{Q(x)\}}$$

図 5 app の推論規則 1

$$\frac{\{L.mem(x) \wedge P\}\ x.G()\ \{Q(x)\}\ \text{ for all } x}{\{P\}\ L.app(G())\ \{L.forall(z|Q(z))\}}$$

$$\frac{\{P\}\ x.G()\ \{L.mem(x) \wedge Q(x)\}\ \text{ for some } x}{\{P\}\ L.app(G())\ \{L.exists(z|Q(z))\}}$$

図 6 app の推論規則 2

て証明可能である．

```
|- !x S L P Q G.
   UNIQ L /\
   DIFF_OBJ_REF_UPD Q G /\
   DIFF_OBJ_UPD_UPD G G /\
   (!S. MEM x L /\ P S ==> Q x (G x S)) /\
   (!S. ~MEM x L /\ P ==> Q x S) ==>
     (P S ==>
      Q x (FOLDL (\s z. G z s) S L))
```

ここで， P, $Q(x)$, $L.mem(x)$, $L.app(G())$ **の意味は，** S **を環境とするとき，それぞれ，**P S, Q x S, MEM x L, FOLDL (z s. G z s) S L **である．この定理は，**2 **節に示した定理を用いて証明することができる．**

5 例題

以下の表明つきプログラムの証明を行う．証明支援系は未実装であり，証明の流れを概略的に示す．

```
{ L.mem(x) && (x.timer>1) && (x!=null) }
L.app(decTimer());
L = L.filter(isAlive());
{ L.mem(x) }
```

まず，ローカル変数に対する代入文の規則を用いる．また，メソッド isAlive() **の展開を行う．**

```
{ L.mem(x) && (x.timer>1) && (x!=null) }
L.app(decTimer());
{ L.filter(z|z.timer>0).mem(x) }
```

ここで，L.filter(z|z.timer>0) **は，**L **の要素のうち，フィールド** timer **の値が正の要素だけを取り出した列である．**z **は論理式の仮引数である．**

次に，filter **と** mem **に関する公理を適用する．**

```
{ L.mem(x) && (x.timer>1) && (x!=null) }
L.app(decTimer());
{ L.mem(x) && (x.timer>0) }
```

次に，app **に関する推論規則を適用する．推論規則の事前条件** P **には** L.mem(x) **が含まれているので，**2 **つ目の前提条件は無効となる．したがって，結論を導くためには，**1 **つ目の前提条件**

```
{ L.mem(x) && (x.timer>1) && (x!=null) }
x.decTimer();
{ L.mem(x) && (x.timer>0) }
```

を証明すればよい．

メソッドを展開し，フィールドに対する代入文の公理を適用すると，以下の検証条件が得られる．

```
L.mem(x) && (x.timer>1) && (x!=null) ==>
L.mem(x) && (x.timer-1>0)
```

これは自然数に関する公理により証明することができる．

なお，逐次的オブジェクト指向言語の公理，推論規則は [3] **を参考にしている．**

6 考察

6.1 条件の保証について

推論規則の 3 **つの条件について，言語レベルで保証する方法について述べる．**

オブジェクト列の単一性については，単一性を維持するように列に対する操作を意味づけすればよい．例えば，列に対する要素の追加 L.add(x) **を次の関数** Add **により定義する．**

```
|- !(x:'a) (L:'a list).
   Add x L = if MEM x L then L else x::L
```

つまり，列にすでに要素が含まれている場合は追加を行わない．このように追加操作を定義すれば，列をどのように操作しても単一性を維持することが可能となる．

残りの 2 つの条件については，基本的には対象となる表明とメソッドがそれらの条件を満たすことを意味論において証明する必要がある．しかし，毎回このような証明を行うことは検証コストの点で非現実的である．したがって，多少制限が強くなっても，言語レベルで自動的に判別できる制限を導入する必要がある．最も簡単な制限法は，繰り返し呼ばれるメソッドの内部で，他のオブジェクトのメソッドを呼ぶことを禁止することである．この制限のもとでは，メソッドが更新する領域は適用されたオブジェクトのフィールドに限られるため，更新領域が排他的となり，互いのオブジェクトの表明に影響しない．また，内部でのメソッドの呼び出しが禁止されるため，第三者のオブジェクトを介しての間接的な影響もない．この制限はコンパイラにより自動的に検査することが可能である．これよりも弱い制限として，オブジェクトのリンクトポロジーに着目する方法がある．つまり，図 7 のようにオブジェクトのリンクが木構造になっていればよい．木構造のもとでは，オブジェクトに対して繰り返しメソッドが呼ばれたとしても，その内部で呼ばれるメソッドの送信先は，子ノードに限定される．したがって，更新される領域は排他的となり，制限が満たされる．トポロジーが木構造になっていることを保証するには，例えば，あるオブジェクトがオブジェクトを生成した後，そのオブジェクトを自分にしかリンクさせない，といったことを検査すればよい．

6.2 有効性について

本稿では，オブジェクト列に対するループ文に着目して，特定条件下で不変表明によらない推論規則を定義した．本質的には，ループの構造を制限することによりループ不変表明が確定し，それを組み込んだ高級な推論規則をつくることができるということである．問題は，この推論規則がアプリケーションの検証にどれだけ有効であるかである．

オブジェクト列に関する単一性という制限に関して

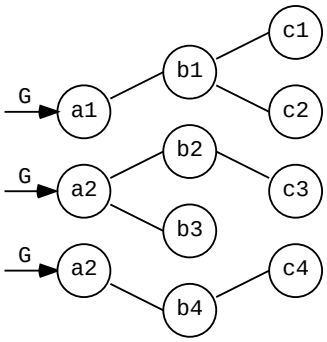


図 7 木構造に対するメソッド呼び出し

であるが，実際のアプリケーションでは，配列をバッグとして使用する，つまり，同一の要素を複数を格納するという頻度は少ないのではないだろうか．少なくとも，集合的に扱うことの方が多いであろう．その意味で，オブジェクト列の単一性という制限は妥当であり，適用範囲を狭めるものではないと考えられる．

メソッドが互いに干渉しないという制限であるが，業務アプリケーションにおいて配列に格納されるオブジェクトは，顧客データや，製品データなどの受動的なオブジェクトが多いのと考えられる．このような受動的なオブジェクトは自らは他のオブジェクトにメッセージを送らない．したがって，このようなオブジェクト列に対するループ文は，前述した，繰り返し呼ばれるメソッドの内部で他のオブジェクトのメソッドを呼ばない，という制限の中に納まり，本稿の推論規則を適用することが可能である．

これらの考察はあくまで予想であり，事例をもとに分析する必要がある．また，事例をもとに，使用頻度の多いループパターンを調査し，それらを効率的に扱える推論規則を定義する必要がある．

7 おわりに

本稿では，逐次的オブジェクト指向プログラムについて，オブジェクト列に対するループが特定条件下で，ループ不変表明によらない規則により推論できることを示し，業務アプリケーションでの適用可能性を示唆した．今後の課題は，定型ループ文とその推論規則を導入したオブジェクト指向言語，表明言語を策定し，検証条件生成器を実装すること，さらに事例研究

により有効性を確認することである．

参考文献

[1] The HOL system.
URL: http://hol.sourceforge.net/.

[2] **林 晋，プログラム検証論，共立出版**, 1995

[3] C. Pierik and F. de Boer, A syntax-directed Hoare logic for object-oriented programming concepts, Technical Report UU-CS-2003-010, Institute of Information and Computing Sciences, Utrecht University, The Netherlands (2003), URL: citeseer.ist.psu.edu/pierik03syntaxdirected.html.