オブジェクト指向プログラムから一般化論理プログラムへの変換

渡辺慎哉、赤間清、宮本衛市

北海道大学工学部

オブジェクト指向の枠組で作成されたプログラムから、GLP の理論をベースとする拡張論理型言語 UL のプログラムへの変換系の作成を試みた。これは、GLP の理論を基礎として、論理型言語や関数型言語などの様々なプログラミング言語を統一的観点から考察する試みの一環である。この変換系が実現することにより、従来、理論的な基盤が弱かったオブジェクト指向の概念を、GLP の理論で把握できるようになる。その結果、GLP 上での他言語との比較を行なったり、プログラム変換を容易にしたりするなどの利点があると考えられる。

Compiling Object–Oriented Programs to Generalized Logic Programs

Shin-ya WATANABE　　Kiyoshi AKAMA　　Eiich MIYAMOTO

Faculty of Engineering, Hokkaido University

N13W8, Kiat-Ku, Sapporo 060, Japan

We have tried to construct a translator which compiles object–oriented programs into programs based on GLP theory. It is a part of the trial that integrates a variety of programming languages onto one general theory.

This translator enables us to understand the concept of object-oriented programming at the GLP point of view. After that, we will be able to compare object-oriented languages with other languages on GLP, or apply the theory of program conversion and type inference to them.

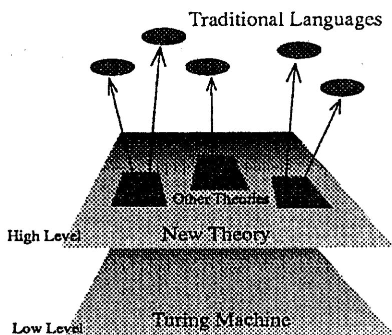

図 1: プログラミング言語の統一的な視点としての新しい理論の必要性

1 はじめに

近年、多種多様なプログラミング言語およびプログラミングパラダイムが、様々な分野で用いられている。このプログラミング言語の多様化に伴い、並行プログラミング言語の分野では、Language Comparison[3] の研究が行われている。言語の比較を行うことにより、あるプログラミング言語が有している特徴が他の言語ではどのような概念として導入されているか、また、言語間に共通した部分や本質的に異なっている部分はどこかを把握することができる。さらには、それが、新しい言語を構築する際の方針になり、また、既存の言語の改良を促すことにもなる。

それを行う手段としては、各言語の基礎となっている理論を比較することが考えられる。また、多くの基礎理論を包含する理論上での、各理論の位置付けを行うことも、有効な方法であろう。ほとんどの言語は Turing Machine を基礎としていることから、Turing Machine 上での比較も考えられるが、現存する種々の言語の高水準化に対応するためには、より高いレベルでの比較が望ましい(図 1)。

しかし、特にオブジェクト指向型言語には厳密な基礎理論が存在していないため、直接の理論上での比較を行うことは困難であろう。従って、Language Comparison の対象としてオブジェクト指向型言語のような理論的基盤が貧弱なものを扱う場合には、広範なプログラミングの概念を扱うことのできる理論に基づいて構築されたプログラミング言語を指標として、既存の様々なプログラミング言語をその言語に変換する試みが有効であると考えられる。

GLP の理論 [1] は Prolog 、クラスを扱う Prolog 、CLP 言語、ユニフィケーショングラマーなどを統一的観点から論じることのできる理論体系である。それは、論理型言語の考察から出発した理論であるが、論理型だけでなく他のいろいろなパラダイムをもつプログラミング言語 [2] の理解にも貢献するものと予想している。また、プログラミング言語 UL は、この GLP の理論に基づいて構築された一種の拡張論理型言語である。

本研究では、既存の様々なプログラミング言語を、GLP の理論という視点から統一的に説明する試みの一環として、オブジェクト指向型言語 Smalltalk-80 をとりあげた。この理由は、Smalltalk-80 が他の言語パラダイムを導入しない純粋なオブジェクト指向型言語であり、オブジェクト指向に本質的にかかわる部分のみを考察の対象としやすいためである。Smalltalk-80 は理論的基盤が弱いため、理論上での対応関係を検討するのは困難である。そこで、Smalltalk-80 のプログラムを UL のプログラムに変換する変換系を構築することを研究方針とした。

本変換系を構築することにより、オブジェクト指向の世界では研究が進んでいないプログラム変換や型推論などの技術が適用可能となる。

2 オブジェクト指向プログラムの性質

変換系構築のためには、対象とするオブジェクト指向プログラムの性質を把握しなければならない。文献 [4] にオブジェクト指向パラダイムの特徴がよくまとめられている。それによるとオブジェクト指向の特徴は擬人化、抽象データ型、および継承であるとされている。擬人化はオブジェクトを、自己 (self) を参照可能であり、外部からの使令に応じて行動をし、状態が変化する実体として捉える。これは、特定の問題—例えばウィンドウのように物理的な『もの』として対象を表現する問題—に対しては非常に相性がよい。しかし、この特徴を形式化して議論することは困難であるとされている。

抽象データ型と継承は、型理論の立場から見た

オブジェクトの特徴である。抽象データ型は、部品化に貢献する情報隠蔽という特徴の他に、メソッド選択の柔軟な方法を与えている。また継承は、コーディングの省力化に貢献する他に、抽象データ型同様、メソッド選択の方向性を与えるものと捉えることができる。

以上のように、オブジェクト指向プログラムの特徴は、そのままでは意味論を構築することが困難であったり、特定の側面のみを切り出して議論されたりしているが、それらを1つの土台の上で統一的に把握する事ができれば、オブジェクト指向プログラムをより深く理解することができると考えられる。

3 両言語の仕様

本節では、本研究で対象とするオブジェクト指向プログラムと一般化論理プログラムの仕様について述べる。

3.1 オブジェクト指向プログラムの仕様

本研究では変換の対象としてSmalltalk-80のサブセットを選択した。実際のSmalltalk-80から削除した部分は、

- メタクラス
- クラスメソッド
- クラス変数、プール変数

である。ただし、本来のクラスメソッドであるnewは、インスタンス生成のプリミティブとして用意している。また、原理的理解を目的としているため、膨大なシステムクラスはほとんど省略している。現在実現しているシステムクラスは、Object, Number, SmallInteger, Boolean, True, False, BlockContext, UndefinedObject, Character, Stringである。これらのクラスにおけるプリミティブメソッドはULのレベルで実現している。

例えば、クラス定義は、

```
Object subclass: #Example
       instanceVariableNames: 'x y'
```

のように表現することにする。また、メソッド定義はカテゴリを省いて、

```
SmallInteger method
  factorial
    (self = 0)
      ifTrue:[↑ 1]
      ifFalse:
        [↑ self*(self-1) factorial].
```

のように表現する。

3.2 一般化論理型言語ULの仕様

オブジェクト指向プログラムの変換先である一般化論理プログラムにはULを用いる。この言語の特徴は、情報付き変数を持ち、変数の情報間のunificationを定義できる所にある。プログラムは節の集合であり、節は論理型言語と同様に、

```
H :- B1, B2, ..., Bn
```

という形をしている。情報付き変数の記法は、

```
<変数名>~<情報>
    where <情報> = 任意のS式
```

とする。例えば、dogという情報を持った変数Xは、X~dogと表記される。この情報付き変数はULに非常に大きな表現力を与えている。

従来の論理型言語はアトムの構造が制限されていたために、unificationの規則はあらかじめ定められている。それに対してULではアトムに任意の構造を許し、その構造に対する解釈も任意である。従って、アトム同士のunificationもプログラムに応じて異なって構わない。そのためULでは、一般化されたアトムのunificationをユーザ定義することが可能となっている。

情報付き変数同士のunificationは、次のように記述する。

```
defUnify(<情報1>, <情報2>, <情報3>)
```

これは、

> 『<情報1>を持った変数と<情報2>を持った変数はunify可能であり、unifyした場合には<情報3>を持った変数となる』

ということを意味する。例えば、それぞれdog及びanimalという情報が付いた変数同士のunificationがdogになる場合の定義は次のようになる。

```
defUnify(dog,animal,dog)
```

従って、X~dog とY~animal とが unify することによって、それぞれが、X~dog という情報に変わることになる。

この情報付き変数と情報同士の unification の定義は、オブジェクト指向プログラムにおける状態変化や継承などを GLP の世界にマッピングするのに威力を発揮する。

4 変換系構築の方針

この節では、オブジェクト指向プログラムの各構成要素が UL の枠組にどのように変換されるかを述べる。本研究で扱う構成要素は、

1. クラス定義
2. メソッド定義
3. メッセージ送信式
4. 変数
5. 擬似変数
6. 代入式

である。第 1 節で述べたように、Smalltalk-80 には厳密な理論的基盤が存在しないため、理論を基にした変換方略を用いることができない。従って、上に挙げた各構成要素が Smalltalk-80 上でどのように機能するかを検討し、それらをそれと等価な機能を実現する UL のプログラムに変換する。上に挙げた構成要素のうち 1–3 は、抽象データ型と継承を実現している。また、4–6 は、擬人化を実現するためのものと考えられる。

4.1 インスタンスオブジェクトの表現

Smalltalk-80 におけるインスタンスオブジェクトは UL では情報付き変数として扱われる。それには、定数オブジェクトも含まれる。情報付き変数に与えられる情報は、それに対応するインスタンスオブジェクトに必要な情報、

- オブジェクトが属するクラスの情報
- オブジェクトが有するインスタンス変数の情報

である。従って Smalltalk-80 のインスタンスオブジェクトは、

```
?~[[class,<className>],
   [<slotName>,<variable>], ... ]
```

という形に変換される。

4.2 クラス定義

メタクラスやクラスメソッドを省略したため、クラス定義は継承関係の把握とインスタンス変数の特定のみに関与する。従って、クラス定義に対応した UL のプログラムは生成されない。単に、インスタンス変数の情報を登録することと、継承構造の情報を登録することに用いられる。インスタンス変数の情報及び継承構造の情報は、それぞれ、

```
slot(<クラス名>,<スロット名>, ... ).
super(<スーパークラス名>,<クラス名>).
```

としてデータベースに登録される。'slot' は、そのクラスにどのようなインスタンス変数があるかを示し、'super' は、そのクラスの直接のスーパークラスは何であるかを示す。

例えば、新たなクラスを、

```
Object subclass: #Example
  instanceVariableNames: 'x y'
```

のように定義した場合、インスタンス変数の情報が、

```
slot(example, x, y).
```

として登録され、継承構造の情報が、

```
super(object, example).
```

として登録される。

4.3 メソッド定義の変換

Smalltalk-80 におけるメソッド定義は、UL のクローズ、

```
H :- B1, B2, ..., Bn
```

に 1 対 1 に変換される。このとき、H はメソッド名を変換したものになり、B1, B2, ..., Bn はメソッド本体を変換したものになる。

例えば、以下のようなメソッド定義、

```
Example method
  test: arg1 nextArg: arg2
    <body of test:>.
```

は、ULでは次のようになる。

```
test(?^[class,example], Ret, Arg1, Arg2)
  :- <translated body of test:>
```

head部は、述語名がメソッド名に対応したものとなり、第1引数がクラスオブジェクト、第2引数がメソッドの返り値を渡すための論理変数、残りがSmalltalk-80の引数に対応する。逐次型のオブジェクト指向言語は、メッセージ送信自体が値を持つ構造であるが、ULでは変数を通して情報の受け渡しをするので、返り値を渡すための第2引数が必要となる。

ここで示したULにおけるメソッドの表現は、CLOSにおけるclassical-methodに対応しており、メソッドの変換形式としては自然なものであるといえる。今回は、Smalltalk-80を対象としているため、第1引数のみをメソッドの特定に関与させているが、CLOSのmultiple-methodへの拡張も容易に行うことができる。

4.4 変数の変換

Smalltalk-80におけるインスタンス変数および一時変数は再代入可能である。従って、prolog等で用いられる通常の論理変数をそのまま用いることはできない。本変換系ではULが持つ情報付き変数を用いて再代入可能な変数を実現している。具体的に言うと、Smalltalk-80の変数は`?^[var]`という情報付き変数に変換され、その変数に`<object>`が代入された場合には、`?^[var,<object>]`となる。また、その変数に`<object'>`が再び代入されると、その変数は、`?^[var,<object'>]`に変化する。すなわち、Smalltalk-80の変数への代入は、通常の論理変数のunificationに変換されるのではなく、情報付き変数に対する情報の付け変えに変換されることになる。

4.5 擬似変数の変換

Smalltalk-80における擬似変数には、self、super、null、true、falseがあるが、これらのうちsuper以外のものは、それらが指すインスタンスオブジェクトを表現する情報付き変数に変換される。その対応表を表1に示す。

表1 擬似変数

Smalltalk-80	UL
self	?^[class,<変換中のクラス>]
null	?^[[class,undefinedObject]]
true	?^[[class,true]]
false	?^[[class,false]]

擬似変数superの変換は表1のように単純にオブジェクトに変換することはできない。なぜなら、superは特定のインスタンスオブジェクトを指すものではなく、現在実行しているメソッドが存在するクラスの直接のスーパークラスを指すものだからである。本変換系では、レシーバオブジェクトに相当する情報付き変数に対して、現在実行中のメソッドがあるクラスのスーパークラスの情報を実行時に付加するような、述語呼び出しの列を生成することにより実現している。生成される述語列は、以下のようなものである。

```
binding0(Obj, [[class,Class]|Slots]),
super(Super, Class),
setr(Obj, [[super,Super],
          [class,Class]|Slots])
```

ここで、述語binding0およびsetrは、それぞれ情報付き変数の情報部分を取り出す述語、および、情報部分を付け替える述語である。

4.6 代入式の変換

Smalltalk-80における代入式は、一般に次のような形をしている。

<変数> ← <式>

これを変換した場合、まず<式>を評価し、その結果得られたオブジェクトを<変数>に代入する述語列を生成しなければならない。また、<変数>も、前節で述べた再代入可能な変数(`?^[var]`)に変換する必要がある。従って、上の代入式は、

```
<<式>を変換した述語列>,
assign(<<変数>を変換した情報付き変数>,
        <<式>を評価した結果得られる値>)
```

という形に変換される。ULの述語assignは、第1引数となる情報付き変数の情報部分を、第2引数の値に置き換えるものである。

Smalltalk-80（に限らず一般の手続き型言語）における破壊代入は、LP(Logic Programming)の理論を逸脱した操作である。しかし、GLPの理論においては、代入を理論から完全に排除することはせず、一種のspecializationとして、理論の枠組の中でとらえようとしている。

4.6.1 メッセージ送信式

メッセージ送信はSmalltalk-80の主要な部分である。Smalltalk-80では、オブジェクトにメッセージを送り、その返り値となるオブジェクトを利用して再び新たなオブジェクトにメッセージを送ることを繰り返して計算が進められる。

Smalltalk-80側の構文は、次のような形をしている。

```
<object> mes1:<arg1> mes2:<arg2> ...
```

このメッセージ送信は<object>のメソッドコールを行なうので、対応するUL側のプログラムは、<object>を第1引数とした述語呼び出しになる。さらに、引数<arg1> <arg2> ... はそれぞれが式であり得るので、式の計算を行なう述語呼び出しの列が生成されていなければならない。従って、上のメッセージ式を変換した結果は次のような複数の述語呼び出しのand結合になる。

```
<arg1 の計算をする述語列>
<arg2 の計算をする述語列>
    ......
methodName(<object>,
           <retVar>,
           arg1, arg2, ... )
```

ここでarg1, arg2, ... は、各引数の計算結果を引き渡すための変数である。例えば、階乗を計算するメッセージ送信、

```
10 factorial
```

は、SmallIntegerクラスのメソッドfactorialをコールするので、対応するULのプログラムも、第1引数がSmallIntegerのクラスオブジェクトである述語factorialを呼び出す。

```
factorial(Obj^[[class,smallInteger],
               [value,10]],
          R01)
```

ここでR01は、計算結果を受け渡しする論理変数である。この述語呼び出しは、SmallIntegerクラスのメソッド定義、

```
factorial(?^[class,smallInteger],
          Ret01)
      :- <body of factorial>
```

のhead部とマッチすることになる。このとき、クラスを識別するオブジェクト

```
?^[class,smallInteger]
```

は、インスタンスオブジェクト

```
Obj^[[class,smallInteger],[value,10]]
```

とunifyしなければならない。本変換系では、この、メソッド定義とメソッド呼び出しの両方に現れる情報付き変数同士のunificationを、ULが持つunification定義機能により実現している。すなわち、オブジェクト指向プログラムにおけるメソッドサーチは、ULにおけるunificationとして意味づけることができるのである。その実際の実現方法は次節で述べる。

また、引数付きのメッセージ送信の例、

```
50 plus: (10 factorial)
```

は、まず、引数となる(10 factorial)の計算を行なう述語を生成し、次にplus:を実行する述語を生成する。実際の変換結果は、以下の通りである。

```
factorial(Obj^[[class,smallInteger],
               [value,10]],
          R01),
plus(Obj^[[class,smallInteger],
          [value,50]],
     R02, R01)
```

この場合、引数の計算結果が返される変数(R01)が、本体の述語呼び出しの引数として利用されている。

4.7 継承の実現

継承を実現することにより、スーパークラスで定義しているメソッドを利用可能となる。このとき、メソッドサーチ機構をインタープリタとして導入することは、変換系の構築という目的からふさわしくない。そこで、本変換系では、ULの特

徴である情報付き変数の unification 定義機能を利用し、インタープリタではなく、unification として継承機構を実現する。その内容は以下の通りである。

```
defUnify([class,ClassA],
         [[class,ClassB]|Slots],
         [[class,ClassB]|Slots])
  :- isa(ClassA,ClassB).

isa(C,C).
isa(CA,CB) :- super(CA,C),isa(C,CB).
```

この定義によって、あるクラスのインスタンスオブジェクトは、自分の属するクラスのメソッド、及び自分の全ての上位クラスのメソッドと unify 可能となる。

4.8 変換例

ここでは、クラス定義・メソッド定義・実行の過程がどのように変換されるかを説明する。

ます、新しいクラス example を定義する。

```
object subclass: #example
  instanceVariableNames: 'x y'
```

この定義によって、インスタンス変数の情報が、

```
slot(example,x,y).
```

という形で登録され、さらに、継承構造の情報が、

```
super(object,example).
```

という形で登録される。この継承構造の情報により、前節で述べた unification の定義から、

```
example インスタンス -> example クラス
example インスタンス -> object クラス
```

という 2 種類の unification が新たに許可されることになる。

次に、example クラスのメソッド定義、

```
example method
  test: arg
    |tmp|
    tmp <- arg + 5 factorial.
    tmp print.
    ↑ self
```

が実行されることにより、次のような UL のプログラムが生成される。

```
test(V9~[class,example],
     V8~[var,V9~[class,example]],
     V7) :-
  factorial(V6~[[class,smallInteger],
                [value,5]],
            V3),
  plus(V7,V2,V3),
  assign(V1~[var],V2),
  print(V1~[var],?).
```

ここで定義されたメソッド test を実行するためには、次のような式を入力すれば良い。

```
example new test:3
```

この式は、以下のように変換される。

```
new(example,R0),
test(R0,V3,V2~[[class,smallInteger],
               [value,3]]).
```

述語 new はプリミティブであり、そのクラスのインスタンス変数のデータベース (slots) を参照し、そのためのスロットを持ったインスタンスオブジェクトを返す。この例では、example クラスのインスタンス、

```
?~[[class,example],
   [x,?~[var]],
   [y,?~[var]]]
```

を第 2 引数 (R0) に返す。この R0 は述語 test の第 1 引数となり、test が実行される。

5 考察

ここでは、本変換系を構築する際に、問題点としてクローズアップされた点に関して考察する。

5.1 式の評価の順序性

Smalltalk-80 では、式の評価の順序が重要である。その直接的な要因は 1 つの変数に対する複数回の代入操作にあることは自明であろう。これは、Smalltalk-80 のメソッドを単一代入にすることによっても解決できない。図 2 のように、共有オブジェクトの存在によって、メッセージ送信にも順

序性が必要となるためである。この順序性は、変換後の UL のプログラムの宣言性を減少させる要因となる。

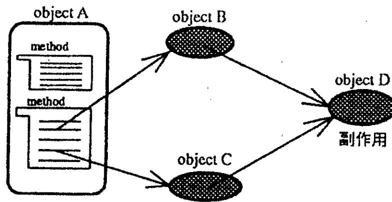

図 2: メッセージ送信の順序依存性

5.2 バックトラック

Smalltalk-80 はバックトラックのない言語であるので、それを変換した UL のプログラムもバックトラックを利用しないものとなっている。しかし、UL はバックトラック機能を備えているので、バックトラック機能を有したオブジェクト指向型言語を考えることができる。

通常、破壊代入を行なう言語にバックトラックを導入した場合、分散シミュレーションにおけるロールバックやアンチメッセージのように、バックトラックを管理する機構を考案する必要がある。しかし、その言語から UL への変換系を構築し、UL のバックトラック機能を利用することにより、破壊代入の自動的なキャンセリングまでが可能となる。これは、本論文で示したように、破壊代入を UL の一般化アトムを用いて実現できることによるところが大きい。

5.3 擬似変数 super の役割

Smalltalk-80 では、あるクラスの直上のクラスを指す擬似変数として super が存在する。擬似変数の変換の節で述べたように、super は他の擬似変数とは性質が異なり、実現方法も不自然で複雑なものになる。

super は、インスタンス変数の初期化など、複数の処理を 1 つのメソッドで実現しようとする時に用いられることが多い。従って、そのメソッドを複数のメソッドに分割し、自分とスーパークラスに分配するようにプログラミングすることにより、super の必要性を減少させることができると考えられる。変換結果の不自然さから考えても、super を用いる必要のない仕様を持ったプログラミング言語が、より好ましい言語であると考えられる。

6 おわりに

本研究では、オブジェクト指向型言語のパラダイムを、GLP の理論の観点から考察することを目的として、Smalltalk-80 のサブセットから UL への変換系を構築した。その結果、擬似変数 super による継承階層上のメソッドサーチの順序性の強要は、UL による継承機構の実現に、強い制限を加えたものであることが分かった。このことにより、メソッドサーチに継承構造上の順序性を求めないオブジェクト指向言語の可能性を考えることができる。また、変数への代入操作は、通常の LP の理論を逸脱した操作であるが、GLP/UL の理論では論理型の延長として旨く捉えられる見通しが得られた。

上記以外の部分、例えば、メッセージ送信とメソッド定義とのマッチングや、メッセージ式同士のオブジェクトの受け渡しなどは、UL の情報付き変数とそれらの unification を用いることにより、無理なく UL のプログラムに変換できた。この親和性の高い部分は、プログラム変換や型推論の適用を行い易い部分であると考えられる。

参考文献

[1] Kiyoshi Akama: Sufficient Conditions of Two Inference Rules for Generalized Logic Programs, Proc of LPC'91, pp.161-170(1991)

[2] 繁田良則, 赤間清, 宮本衛市: 関数型言語から論理型言語への変換について, 日本ソフトウェア科学会第 8 会大会論文集,(1991)

[3] E. Shapiro: Separating Concurrent Languages with Categories of Language Embeddings, Technical Report CS91-05, The Weizmann Institute of Science, 1991

[4] S. Danforth, C. Tomlinson: Type Theories and Object-Oriented Programming, ACM Computing Surveys, Vol. 20, No. 1, pp. 29-72(1988)