2H-5　オブジェクト指向言語ＥＳＰにおけるタイプチェックシステムの試作

小中裕喜、青柳龍也、田中英彦
(東京大学工学部)

1．はじめに

1．1　タイプチェックシステムのねらい

論理型言語の変数にはタイプがないといわれるが、通常のプログラミングでは人間は変数のタイプを暗黙のうちに意識している。また組み込み述語には引数のタイプが厳密に規定されているものが多い。これらのことから、タイプを静的に解析してデバッグなどに応用することが考えられる。そこで、実用的なレベルでこのような面をサポートしたプログラミング環境を提供しようとするのが本システムである。

1．2　現実的なシステムへのアプローチ

今回我々が目指しているものは現実的なシステムであり、プログラミング時に静的に１つでも多くのエラーをチェックしてくれるものである。従ってタイプ推論・チェックアルゴリズムの正当性は重視しない。むしろシステムがチェックしきれない部分は、警告を出してユーザの指示を待つといったような対話的なシステムである。

1．3　システムの概要

今回試作しているシステムは逐次型推論マシンＰＳＩ上の論理型オブジェクト指向言語ＥＳＰを対象としている。

システムはメソッドやスロットのタイプの仮定やインヘリタンスの関係などを保持するライブラリをもつ。ライブラリは３つに分類される。ＰＳＩの機械命令ＫＬ０の提供する組み込み述語に関するもの、オペレーティングシステムＳＩＭＰＯＳの提供するクラスに関するもの、そしてユーザがすでに定義したクラスに関するものである。その構成を図１に示す。

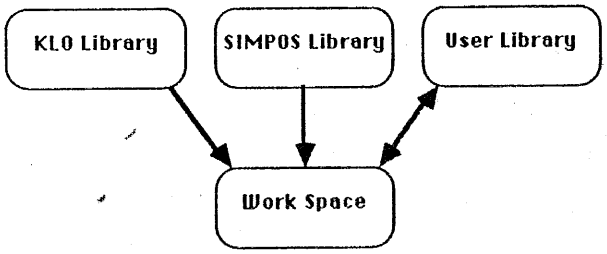

図１　システムライブラリの構成

システムはエディタと密に結合されている。そしてユーザがエディタの中で述語の記述を行ってからあるコマンドを入力することにより、システムが起動される。その際システムに登録されていない述語がでてくれば、警告を出してユーザに指示をあおぐ。

またいくつかのクラス定義を一斉にタイプチェックすることも可能であり、その結果を登録することによってuserLibrary は随時更新されていく。

2．ＥＳＰのタイプチェック

2．1　対象とする言語

ＥＳＰはPrologにオブジェクト指向を導入したものである。その変数にはPrologの論理変数と、副作用を持つスロットの２種類がある。以下では変数と呼ぶ場合、前者をさすものとする。

クラス内に記述される述語は、メッセージのプロトコルとなるクラスメソッド、インスタンスメソッドに関しては、その主処理述語とそれらの前処理・後処理を表すbefore・after デーモン、そしてクラス内部でのみ参照されるローカル述語、スロット初期化述語に分類されるが、スロット初期化述語は現在扱っていない。またスロットに関してはタイプ推論は行わず、ユーザから入力されたタイプをチェックするにとどめている。recursionはローカル述語にのみ許し、mutual recursionは現在のところ扱っていない。

2．2　タイプとモード

今回のタイプチェックシステムはメソッドの引数のタイプとモード、及びスロットのタイプを推論・チェックするものである。

以下にタイプの定義を示す。

```
PrimitiveType ::= atom|int|float|#classname
Type ::= PrimitiveType|string|
         stack_vector|heap_vector|
         α|$(α)|Type;Type|not_referred
```

「α」はクラスの要素を持つタイプ変数である。また「$」はクラスのinstanciate を、「;」はタイプのユニオンを表す構成子である。特に、

```
number = int;float
atomic = atom;number
heap = string;heap_vector;#classname;$(#classname)
any = atomic;stack_vector;heap
```

という表記を便宜上用いている。

ロケーション、コード、保護付きデータについては扱わない。またＥＳＰではスロットのタイプはheapかatomicに限られている。

本システムでは、構造体の中身のタイプまでは追求せず、その要素のアクセスなどで必要があればユーザにその都度タイプの入力の入力を要求する。しかしスタックベクタに関してはユーザがタイプを定義することによってその中身を推論・チェックすることも可能となる。その定義例を以下に示す。

Type Check System on Object Oriented Language ESP
Hiroki KONAKA,Tatsuya AOYAGI,Hidehiko TANAKA
Univ. of Tokyo

```
TypeDef : list(Type) = [];[Type|list(Type)]
```

述語の引数についてはモードも導入している。モードは大別すると、述語の実行前に未定義かどうか、実行後に未定義かどうかで４種類存在するが、現在は実行後に未定義なものは扱わず、実行前に未定義であるものを入力モード、定義済みであるものを出力モードとしている。また中間的なもの（例えば [3,A]）は扱っていない。出力モードはタイプの前に「＾」をつけて表す。

述語のタイプ（及びモード）は述語名の後に引数のタイプ及びモードを列挙したもので表す。その例を組み込み述語add の場合について示す。

```
add(int,int,^int); add(float,float,^float)
```

２．３　対象とするエラー

本タイプチェックシステムが検出できるエラーは以下に示すとおりである。

- 未定義入力（モードエラー）
- 定義済み出力（モードエラー）
- 不正入力（タイプエラー）
- データ型不整合（タイプエラー）
- メソッド未定義
- スロット未定義

２．４　タイプ推論・チェックの概略

ある述語のタイプを推論・チェックする基本的なアルゴリズムを説明する。述語の定義は通常いくつかのクローズからなるので、そのタイプはクローズごとに決定されたタイプのユニオンとなる。またメソッドの場合はさらにデーモンやインヘリタンスを考慮する。

クラス主処理述語内のクローズのタイプを決定する場合を示す。まずボディB_iに対応した、Libraryに存在する（インヘリタンスも考慮した）述語の全てのタイプのユニオンをD_iとする。not_referredを考えないと引数に現れる変数のタイプはそれが最初に現れるボディB_jで決まり（但し第１引数はそのクラスかサブクラスのオブジェクトでなければならないという条件がある）、以後は同タイプで入力モードで参照されなければならない。

引数に現れる変数の一部が入力モードで各々あるタイプを持つ場合は、それぞれが最初に現れるボディB_kに対応するD_kの内、それらを受け付ける可能性のあるものだけを新たにD_kとしておく。そして$D_1, D_2 \ldots$のユニオンを展開し、その任意の組合せの集合をＤＳとする。

ボディにのみ現れる変数は最初に出力モードでタイプが返され、以後は同タイプ、入力モードで参照されなければならない。ＤＳの全要素について各ボディとマッチングをとって変数についての上の条件を満たすかどうかを調べる。出力モードで返される可能なすべてのタイプについて成功した結果が得られれば、その結果からクローズのタイプを定める。それ以外は失敗とみなす。

引数が任意のタイプの場合について上記を行う。すべて失敗すればそのクローズはタイプ・モードエラーを起こしていることになり、それ以外は得られた結果のユニオンをそのタイプとする。

ここで親クラス#ex0、子クラス#ex2をもつクラス#ex1のクラス主処理述語内のクローズのタイプを決定する場合を例示する。なおボディのタイプはすべて既知とする。

```
:msg0(Self,A,B) :- :msg1(Self,A),:msg2(#ex3,A,B);

Types in Library
class methods in ex0 : msg0(#ex0,^float,^float)
              in ex1 : msg1(#ex1,^int);
                       msg1(#ex1,^float)
              in ex2 : msg1(#ex1,^int);
                       msg1(#ex1,^float)
                       msg1(#ex2,^int)
              in ex3 : msg2(#ex3,int,^int)
```

まずSelfは#ex1か#ex2でなければならない。しかし#ex1とするとmsg1によってA にintかfloatのどちらが返ってくるかわからないので、msg2でfloat が受け付けられないことにより、失敗する。

ところがSelfが#ex2の時は成功する。このときこのクローズのタイプはmsg0(#ex2,^int,^int)となる。

親クラスにも同じメソッドがあり、実行時これらはOR結合されるので、#ex1のクラスメソッドmsg0のタイプは

```
msg0(#ex0,^float,^float);msg0(#ex2,^int,^int)
```

となる。

３．　今後の課題

３．１　エラーの根元の特定

本タイプシステムではタイプエラーやモードエラーが生じた場合、一般にその原因となった箇所を特定できない。しかしなんらかのアルゴリズミックな手法によって原因が（半自動的に）わかると、さらにプログラミングの効率は向上する。この点については現在検討中である。

３．２　ライブラリの更新

userLibrary に新たなクラスを登録する際、既存の情報も変更が必要となる場合がある。基本的には全クラスに関してタイプチェックを行えばよいのであるが、非能率的である。そこでインクリメンタルで効率よい更新法の採用が必要となるが、この点に関しても検討中である。

参考文献

[1] R.Milner,"A Theory of Type Polymorphism in Programming",JCSS17,pp.348-375,1978

[2] P.Mishra,"Towards a theory of types in Prolog",Proc.IEEE Internet.Symp.Logic Programming,1984

[3] N.Suzuki,"Inferring Types in Smalltalk",Proc.ACM 8th POPL,1981

[4] 青柳他,"オブジェクト指向言語における名前づけとタイプ",情報処理学会第３５回全国大会,3R-8,1987

[5] T.Kanamori and K.Horiuchi,"Type Inference in Prolog and Its Applications",ICOT Technical Report,TR-095,1984

[6] A.Mycroft and R.A.O'Keefe,"A Polymorphic Type System for Prolog",Artificial Intelligence 23,pp.295-307,1984