程序推导的表示

吕建国　徐家福　（南京大学计算机软件研究所）

摘　要

Formal Program development deals with three aspects, specification, program, and the development from specification to the program. Usually they are denoted in different notations, which hampers the formal construction of programs, especially the manipulation of program development. This paper proposes an approach to express the three elements in a single logic framework, which is anticipated to facilitate analogical programming.

软件自动化是提高软件生产率的关键途径之一[1]，而类比程序设计是软件自动化的一条重要途径[12]，它的主要思想是系统地将求解一个问题的程序转化/修改为求解另一个相似但不等同的问题程序。为达到此目标，现在的共识是充分利用从规格说明到程序的推导，即所谓推导类比方法[3]。当前，程序推导的形式化表示成为该领域研究的焦点。

在定理证明领域，证明过程的表示早在1968年de Bruijn[2]的工作中得到了充分重视。在他所主持的AUTOMATH项目中，数学定理证明过程可用一种带类型λ演算的项来表示。尔后，历经二十余年的孕育发展，开展了不少重要工作，比如美国Cornell大学的交互式数学证明构作系统NuPRL[4]，英国Edinburgh大学的逻辑框架(ELF)[3,8]，瑞典Martin-löf的直觉主义类型论[11]和北航的逻辑框架BLF[14]。这些工作主要根据的是"命题作为类型"原理[10]：命题可看作是一个类型，命题为真当且仅当该类型中有元素，命题的证明便是该类型中的一个对象。这样做的一个直接好处是证明的验证过程可化归为类型检查过程——p是A的一个证明当且仅当p具有类型A，从而为证明构造系统和类型检查系统提供了一个坚实的逻辑基础。

遗憾的是，现有工作是在较为广泛的范围内讨论这一问题的，没有具体针对程序构造这一特定领域，离应用于程序构造系统的实现尚有距离。本文试图在一个带类型的λ演算框架下表示规格说明、程序以及从规格说明到程序的推导；即程序推导是该演算中的λ项，而规格说明和程序是该演算中的类型。如果一个λ项d具有类型p，这里p表示一个程序，则d是p的一个正确推导。这样，推导的正确性证明化归为类型检查问题，这项工作已有深入研究。另一方面，程序推导的这种形式化表示有利于程序推导作形式处理。下面顺次介绍该演算的抽象语法、表达能力及其语义，最后与相关工作作一比较。

一、形式系统

在带类型的λ演算的项和自然演绎之间有着意想不到的类似关系。具体说来，我们可以把类型对应于命题[10]，项对应于自然演绎[8,10]。既然程序构造可看作构造性定理证明过程，λ演算的项和程序推导过程之间也应存在着某种对应。下面给出一个扩充了

的带类型的λ演算，以讨论这种对应关系。

按照de Bruijn[5]，这里有一个与普通λ演算有所不同的观点，即不区分类型和对象，同一个项既可用作类型也可以用作对象。项的所在环境决定其所起的作用。下面给出项的抽象语法：

项::=空
| a, b, c, …
| x, y, z, …
| A, B, C, …
| λx:A•B
| AB | B\A
| ‹A, B› | spread (‹A, B›, λxλy•C)
| A..B

这里x, y, z, …是变元符号，a, b, c…是常元符号，A, B, C…的取值范围是项。

对以上的抽象语法作说明如下：

空项是不可见项。它的引入使得项的范围扩大了。与普通λ演算相比，我们有更多的合式项，比如λx:A，(λx:A)B等。

λx:A•B称为抽象。与众不同的是，它既表示函数，又表示推理规则。当A是一个对象集合，B是函数体时，λx:A•B是一个函数抽象；当A、B是命题时，按照命题作为类型原理，λx:A•B恰恰表示了对命题A的任意一个证明x，我们可以得到B的证明，因而λx:A•B表示在假设A下B成立。由于我们有时并不关心x的具体值，因而采用{A}B或{x:A}B这样的记号来表示抽象。

A、B的并置称为作用。B\A是作用的另一种表示，它定义为B\A=AB。采用这样的记号是为了使得推导的表示符合习惯，这在后面有所说明。

‹A, B›是元组，spread (‹A, B›, λxλy•C) 提供了处理元组的手段，其含义为：

$$\text{spread}(\langle A, B\rangle, \lambda x\lambda y\cdot C) = C\Big|^{(A,\ B)}_{(x,\ y)}$$

这里$A\Big|^{a}_{x}$表示将A中x的出现置换为a。置换的定义同普通λ演算。

特别是投影操作可表示为：

first (‹A, B›) =spread (‹A, B›, λxλy•x),
second (‹A, B›) =spread (‹A, B›, λxλy•y)。

A..B称作判断，它断言A具有类型B。

子项的定义同普通λ演算。

这样的演算可以表示程序推导中的以下各种概念。

1.说明和假设 说明变元x具有类型A可以表示为λx:A。如果A是一个命题，那么假设A成立可记为λx:A，其直觉含义是命题A具有证明x。为符合习惯，以下直接将类型说明λx:A记为x:A，将假设A成立(λx:A)记为A。

2.定义 函数定义在程序中必不可少。x定义为A可表示为A\(λx:B)，这里B是一个与A的类型可变换的类型，其定义见语义部分。该表示的直觉含义是与其右侧并置的项中的所有x的出现将被置换为A，为简单计，以下记为x:=A。

3.命题 假设我们有两个原始类型o,l，分别表示命题的类和对象的类[8,9]。那么A是一个命题可表示为A:o，A是一个对象表示为A:l，A蕴涵B可表示为λx:A•B，即{A}B。

4.演绎和函数 用λ项表示函数是大家熟知的，比如{x:int}x^2表示了求整数平方这么一个函数。在A的假设下B成立可表示为{A}B。比如等关系的传递规则可粗略地表示为{x=y, y=z}x=z。

5.多形态 函数的多形态是指一函数的定义域或值域不限于一个类型。假设我们有一个常元sort，它是一个类型的类，则通过它我们可以引入类型变元，从而表达多形态函数。比如适用于多种类型的恒同函数可表示为：{t:sort} {x:t}x。

6.规格说明，程序和推导 这在下一节中作详细讨论。

二、例

1.自然数类型

与抽象数据类型的代数规格说明相似，我们可以表示自然数类型如下：

```
nat:sort;
o:nat;
s:{nat}nat;
1 :s(0);
+:{nat, nat}nat;
add:({x, nat}x+0=x,
    {x, y:nat}(x+s(y))=s(x+y))
```

2. 程序和规格说明

这里的规格说明指以一阶逻辑公式书写的前后断言或清晰但效率不高的函数式程序[6]；程序限于带递归等式的函数式程序。上节已讨论了该演算中逻辑公式的表示，这里说明函数式程序表示，比如求阶乘的程序可表示为：

```
fac:{nat}nat;
dfac: fac(1)=1,
      {n:nat}fac(n+1)=(n+1)*fac(n)).
```

3. 程序推导规则

我们以经典的fold/unfold程序变换模式为例[5]。这里，程序推导规则主要是展开、卷叠、定义、定律作用等。这些规则均可说明为演算中的项，比如展开规则可说明如下：

```
unfold:{t, s:sort;
        x, y:t; z:s;
        F:{t}s}
        {x=y} ({z=F(x)}z=F(y))
```

4. 程序推导

在Fold/Unfold程序变换方法中，结构清晰但效率不高的函数式程序被转换为效率较高的程序。比如上述阶乘程序的变换可表示为：

```
fac:{nat}nat;
dfac:<fac(1)=1;
     {n:nat}fac(n+1)= (n+1) *fac(n)>;
F:={n, acc:nat}acc*fac(n);
base-dev:=refl (F(1, acc)) .. F (1, acc )
                              =acc*fac(1)
    \unfold (first(dfac)) ..F(1, acc) =
                                  acc*1
    \Law (a*1=a) ..F(1, acc)=acc
recur-dev:={n:nat}
    refl (F(n+1), acc)..F(n+1, acc)= acc*
                                  fac(n+1)
    \unfold (second(dfac)) ..F(n+1, acc)=
                          acc* ((n+1*fac(n))
    \law (a*(b*c)=(a*b)*c)..F(n+1, acc)
                  = (acc*(n+1)) *fac(n))
    \fold(F)        ..F(n+1, acc) =F (n,
                                  acc*(n+1))
```

这里F是新引入的辅助函数。

refl是等关系中的自反律：refl:{t:sort, x:t}x=x。

显然fac(n)=F(n,1)，这里F是效率较高的程序。

三、语义

为定义前面描述的结构的含义，我们粗略地给出该演算的操作语义。

1.类型、归纳和变换

类型函数Typ定义为：

Typ(x)=t　　如果x的类型被说明为t

Typ(λx:A·B)=λx:A·Typ(B)

Typ(AB)=typ(A)B

Typ(<A, B>)=<TypA, TypB>

Typ(A..B)=Typ(A)。

演算中的归约可看作β归约的一种扩展，项t_1归约为t_2记为$t_1 \triangleright t_2$。一些典型的归约规则为：

$$(\lambda x{:}A\cdot B)C \triangleright B\begin{matrix}C\\X\end{matrix}$$

$$A..B \quad \triangleright A$$

$$\text{spread}\ (\langle A, B\rangle, \lambda x\lambda y\cdot C) \quad \triangleright C\begin{matrix}(A, B)\\(x, y)\end{matrix}$$

下面给出一些归约的例子：

```
({n:nat}f(n+1)=(n+1)*f(n)) 2
  ▷f(2+1)=(2+1)*f(2)
```

{x, y:nat} ({x=y}y=x) (0+z)z

▷{y:nat} ({0+z=y}y=0+z) z

▷{0+z=z}z=0+z

归约关系的传递、自反、对称闭包称为变换，记为～。

2. 有效性

有些合式λ项是无意义的，比如λx:nat λy:nat x(y)。由λ项所表达的推导有时是不正确的，比如在上节例子中若对推导作些变动，refl (F(1,acc)) ..F(1,acc)=fac(1) 便不是正确的推导。为保证推导的正确性，我们定义项的有效性如下：

项T是有效的，如果

1) 其所有子项BA满足条件：存在C、D，使得①Typ(B)存在且Typ(B)～λy:C.D

②Typ(A)存在且Typ(A)～C

2) 对所有子项A..B，有Typ(A)～B.

不难验证，上面所举的两个例子均不是有效项。

四、结束语

类比程序设计的现有工作都采用了某种方式来表示程序推导，如Ulrich[13]等使用非形式的程序正确性证明、Dershowitz[6]在程序代码中逐行加注释、Harandi[7]等采用树结构，所有这些表示方式均未能完全形式化。与ELF、BLF相比，主要区别在于：①这里不区分类型和对象。②采用更为丰富的记号以接近实际的程序开发。

在类型论中表示程序推导过程有两大优点：一是将程序推导的正确性验证化归为类型检查问题，为程序推导的构作提供一个良好的逻辑基础；二是有利于对程序推导作形式变换，以实现程序推导的重用，从而达到使用类比推理技术实现程序自动生成的目标。本文在这方面作了一点尝试，进一步的工作是在此表示的基础上，对程序推导作变换。

参考文献

[1] 徐家福，软件笔谈，计算机科学№3，1990

[2] N. G. de Bruijn, A survey of the project AUTOMATH,in Seldin (ed.) To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, 1980

[3] J. Carbonell, Derivationel Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition, in Michalski (ed) Machine Intelligence 1986

[4] R. Constable et al., Implementing Mathematics with Nuprl Proof Development System, Prentice Hall, 1985

[5] R. M. Burstall, A Transformation System for Developing Recursive programs, JACM24(1), 1977

[6] Dershowitz,Programming by Analogy, in Machine Learning, vol(2), 1986

[7] Harandi, Program Derivation Using Analogy, IJCAI'89

[8] R. Harper et al., A Framework for Defining Logics, Proc. of the 2nd Symposium on Logic in Computer Sci., IEEE, 1986

[9] R. Harper et al., Foundations of Programming: Aspects of Research in Ergo, CMU Internal Report, 1990

[10] W. A. Howard, The Formula as Types Notion of Construction, in Seldin (ed)To H. B. Curry: Essays on combinatory Logic, Lambda Calculus, and Formalism, Prentice Hall, 1980

[11] P. Martin-löf, Constructive Mathematics and Computer Programming, in Logic, Methodology and Philosophy of Science, North-Holland, 1982

[12] Scherlis, D. Scott, Inferential Programming, IFIP'83

[13] J. W. Ulrich et al., Program Synthesis by Analogy, Internal Report, 1977

[14] Wi Li, A type-theoretic approach for program development, Future Generation Computer Systems 6(1990)