计算机科学 1998 Vol.25 №.3

复合抽象数据类型结构化规约的形式语义及其模型*)

The Structured Semantics and The Formal Model for Composed Abstract Datatypes

蔡家楣
(浙江工业大学信息工程学院 杭州 310014)

摘 要 本文用构造型说明法讨论了满足结构化要求的各种复合抽象数据类型应具备的语义特性。然后讨论了基于类型理论的ECC扩展构造逻辑和函数式语言两种方式下的模型。

关键词 复合抽象数据类型,语义,构造型说明法,ECC扩展构造逻辑,函数式语言

抽象数据类型ADT是一个涉及数据结构的概念,一个抽象数据类型把一个数据结构及其操作作为一个整体来研究。随着面向对象的软件开发方法的兴起,人们把表示抽象数据类型的说明方法应用于面向对象的设计过程,收到了良好的效果。对于需要进行分步方式提纯开发的大型程序来说,人们希望用具有松散语义的抽象数据类型规约来表示它们,于是仅对程序进行传统意义上的代数语义研究已显不足,研究各种抽象数据类型规约及其集成也就具有了实际意义。我们所开展的研究包括:用结构化的方式来描述一个抽象数据类型的规约语义,即如何用一些基本的抽象数据类型来复合成新的数据类型,直至把任意复杂的对象用一些循环递归的基本数据类型来表示。在这种复合中,一个数据类型的操作既可以是基本数据值,也可以是一个包含复合数据或操作集的抽象数据类型,甚至是抽象数据类型的集合。在我们的课题中所采用的是一种属于构造型说明的算法说明法,用它确定各种复合抽象数据类型的语义,然后分别用函数式语言ML和类型理论ECC为它们建立了实现模型,这些模型将很容易转化为其它程序形式。

1 原子规约、标准规约和复合规约

1.1 构造型规约

目前,说明程序规约的方法基本有三种:操作法、代数法和构造说明法[1]。通常,一个数据类型可以看成为一个多类子代数(multisorted algebra)。对每个类子(sort)而言,有一个该类子的载体集(carrier set)。而该代数则由这些载体集和其上的操作集组成。

所谓操作法规约一般是用强制性编程语言来描述的,如FORTRAN等。载体集是该语言中的数据,而操作则被定义成函数过程。这种规约由于其强制特性,即机器依赖性,无法简洁地表现客观对象,也存在着证明上的困难。

代数法规约本质上是由一个一阶谓词逻辑的公式集组成的,如一组等式或Horn子句等。这种说明方法虽然简洁,但十分抽象,且难以实现结构化。

目前认为较好的是构造说明法,它用一种构造型方法来定义载体和操作。在规约中,载体集被定义为一个形式语言的等价类,而操作则被定义为递归程序。这种说明法在结构化特性和抽象特性两方面都有较好的表现。它的抽象特性来自把某种描述性语言和一个形式机制结合,其载体和操作的表示和所使用的语言的具体语法无关,从而使构造说明法更为接近一个原始代数并且容易实现程序证明。

1.2 原子规约的语义

我们把一个基本抽象数据类型的规约叫做原子规约,如果我们用Σ代数作为构造说明法的形式机制,那么有:

*)本项目得到国家高技术研究发展计划863—306智能计算机主题和浙江省自然科学基金的资助。蔡家楣 副教授,主要研究方向为软件工程和函数式语言。

定义 1 一个基本抽象数据类型的原子规约的基调可以简单地表示为 $\Sigma = (S,O)$，其中，$S = \{S_i | i \in I\}$ 为一个有限集，I 是一个有限下标集，每个 S_i 是一个类子，$O = \{O_j | j \in J\}$ 为另一个有限集，J 也是一个有限下标集，每个 O_j 是一个操作。一个 k 目操作 $O_j (k \geqslant 0)$ 可表示为：

$$O_j : S_1 \times S_2 \times \cdots \times S_k \to S_{k+1}$$[3]

$S_1 \times S_2 \times \cdots \times S_k \to S_{k+1}$ 叫作该操作的算术，或叫全秩。操作集是从载体集中去获取自变元和值的，所谓载体集是指用这个数据类型中涉及的类子的某几个操作，即由所谓构造子(constructor)来生成的一种项语言。

定义 2 如果有 $\Sigma = (S,O)$，让 x 为 S 上的变量集，则关于某个类子 σ 的项 $\Sigma(x)$ 为：(i) 变量 $x_\sigma \in S_i$；(ii) 操作 $OP: \to \sigma \quad OP \in O$；(iii) 操作 $F: \sigma_1 \times \sigma_2 \times \cdots \times \sigma_n \to \sigma \ (n \geqslant 1)$ 记作 $F(t_1, t_2, \cdots, t_n)$，而 $t_1, t_2, \cdots, t_n$ 分别是类子 $\sigma_1, \sigma_2, \cdots, \sigma_n$ 的 $\Sigma(x)$ 项。

这样，具有类子集 S 的基调为 Σ 的一个代数系统 A 就包含了两个部分：

(1) 一个关于每个 $\sigma \in S$ 的类子的载体集 σ^A；

(2) 一个来自 Σ 中操作名为 $F: \sigma_1 \times \sigma_2 \times \cdots \times \sigma_n \to \sigma \ (n \geqslant 0)$ 的操作的完全函数 $F^A: \sigma_1^A \times \sigma_2^A \times \cdots \times \sigma_n^A \to \sigma^A \ (n \geqslant 0)$ 的集。

这样构成的规约叫做一个原子规约。这一规约中所有的载体和操作都来自本规约所定义的基调 $\Sigma = (S,O)$ 或者系统内嵌的基本规约定义的类子的载体和操作集，于是具有较好的模块化结构特性。

1.3 标准规约和复合规约

仅使用原子规约来描述高度复杂的客观世界显然是远远不够的，于是我们希望从一些已有的规约出发，构成任意复杂的规约。通过不同的结构来对规约进行诸如组合、合并、更名、"忘记"部分类子或操作和产生子代数系统或商代数系统的规约等。这样产生的规约叫复合规约。为了形成复合规约，仅对一个代数的数据和操作进行说明是不够的，更重要的工作是扩充一个代数。也就是说，建立一种规约操作，把它作为映射一个输入代数到输出代数的算符来看待。如果把输入代数定义为一个标准代数，它只包含了某些类子和操作，当把某个规约操作施用于这个输入代数时就产生一个扩充的包含了附加输出类子或操作的代数，它可以成为一个新的标准代数。这样，一个任意复杂的代数就可以由循环递归的嵌套的标准代数的规约来表示，形成一种自然的层次结构，它使我们联想到面向对象设计中的类层次。因此，这种复合规约的构成对于解决面向对象的应用是很有利的。

定义 3 如果全部 Σ 代数的类记作 $Alg\Sigma$，输入规约的基调为 Σ_i，输出规约的基调为 Σ_o，一个复合规约的基调应被看作序偶 (Σ_i, Σ_o)。其中，由 $\Sigma_i \cap \Sigma_o$ 得到的种类叫继承。一个规约操作 ρ 有：

$$\rho: Alg\Sigma_i \to Alg\Sigma_o$$

如果表示输入规约 SP' 到输出规约 SP 的映射，则记作 $\rho: SP \to_o SP'$。这一映射也可以是多元的，这样上式就扩展为：

$$\rho: (Alg\Sigma_{i_1}, Alg\Sigma_{i_2}, \cdots, Alg\Sigma_{i_n}) \to Alg\Sigma_o$$

2 复合规约的结构化语义

一个复合规约的基调应被看作序偶 (Σ_i, Σ_o)，但是不同的复合抽象数据类型可以有不同的语义。如果 m_1, m_2 是两个标准代数的规约，S(m) 是复合规约，A 是有基调 S(m) 的代数系统类中的一个，有 $A \in AlgS(m)$，m 为 A 的语义，用 $m(m)(A)$ 表示。

2.1 合并规约 $m_1 + m_2$

规约 $m_1 + m_2$ 的基调为 $S(m_1 + m_2) = S(m_1) \cup S(m_2)$，其上下文条件是：(i) $S_o(m_1) \cap S_o(m_2) \subseteq S_i(m_1) \cap S_i(m_2)$；(ii) $S_o(m_1) \cap S_i(m_2) \subseteq S_i(m_1)$；(iii) $S_o(m_2) \cap S_i(m_1) \subseteq S_i(m_1)$。于是 $m(m_1 + m_2)(A) = m(m_1)(A|S_i(m_1)) \cup m(m_2)(A|S_i(m_2))$。这意味着，$m_1 + m_2$ 规约的输入基调是 m_1 和 m_2 输入基调的并集，且其输出基调也一样。而上下文条件则意味着，如果某个类子或操作在 m_1 和 m_2 的输出基调中都发生，那么它在这两个规约中必须有相同的含义以避免出现两义性。$m_1 + m_2$ 的逻辑含义如图 1。

图 1 中符号 a,b,…,f,g 表示一个类子或操作，进入方框的箭头表示输入类子或操作，离开方框的箭头表示输出类子或操作，虚线表示继承(下同)。

2.2 组合规约 $m_1 \cdot m_2$

规约 $m_1 \cdot m_2$ 的基调为 $S(m_1 \cdot m_2) = ((S_i(m_2), S_o(m_1))$ 其上下文条件是：(i) $S_o(m_2) = S_i(m_1)$；(ii) $S_i(m_2) \cap S_i(m_1) \subseteq S_i(m_1)$，其代数系统 A 的语义为：

$$m(m_1 \cdot m_2)(A) = m(m_1)(m(m_2)(A))$$

该规约 $m_1 \cdot m_2$ 的输出基调是 m_1 的，输入基调是 m_2 的，而上下文条件则意味着 m_1 的输入是 m_2 的输出。

某些由 m_1 输出并由 m_2 输入的类子或操作则是由 m_1 和 m_2 继承的。$m_1 \circ m_2$ 的逻辑含义如图 2。

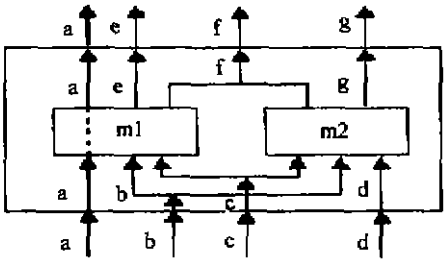

图 1 m1+m2 的逻辑含义

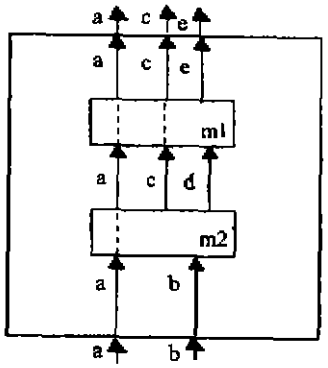

图 2 m1∘m2 的逻辑含义

2.3 遗忘规约 m forget lso

如果 m 是一个规约，lso 是一个类子和操作的列表，那么规约(m forget lso) 的基调是 S(m forget lso) = $(S_i(m), S_e(m)\backslash lso)$。在这里 $S_e(m)\backslash lso$ 是一个去除了 lso 的 m 的输出基调。这样，forget 结构允许"遗忘"某些输出类子或操作。要注意的是，遗忘一个类子意味着同时遗忘所有和该类子有关的操作，它的逻辑含义如图 3(假定遗忘 a,c)。

具有遗忘结构的代数系统的语义是：

$$m(\text{m forget lso})(A) = m(m)(A)|S_e(m)\backslash lso$$

2.4 输出改名规约 m e_rename lso_1 as lso_2

如果 m 是一个规约，lso_1 as lso_2 是类子和操作表。那么，规约 m e_rename lso_1 as lso_2 的基调是 S(m e rename lso_1 as lso_2) = $(S_i(m), \rho(S_e(m)))$，ρ 是一个在 $S_e(m)$ 基调上的改名序偶 (lso_1, lso_2)，即调用 ρ 就可以用 (lso_1, lso_2) 来改变 $S_e(m)$ 的基调，其条件为：(i)ρ 被加进 $S_e(m)$ 中去；(ii)lso_2 中应该没有一个类子或操作来自 $S_i(m)$ 以避免同输入名混淆。这一规约可以改名输出类子和操作，其代数系统的语义是：

$$m(\text{m e rename } lso_1 \text{ as } lso_2)(A) = m(m)(A)|\rho^{-1}$$

即(A 关于 ρ 的商集)，其逻辑含义见图 4。

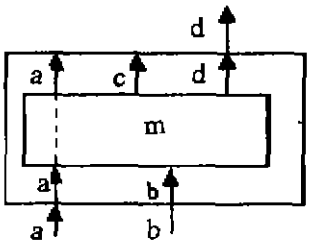

图 3 forget 的逻辑含义

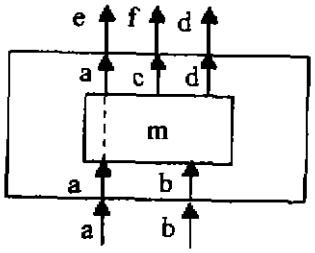

图 4 e-rename 规约的逻辑含义

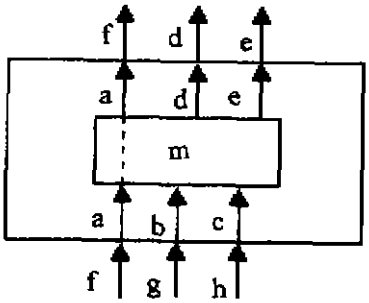

图 5 i-rename 的逻辑含义

2.5 输入改名规约 m i rename lso_1 as lso_2

如果 m 是一个规约，lso_1 as lso_2 是类子和操作表。那么，规约 m i_rename lso_1 as lso_2 的基调是 S(m i rename lso_1 as lso_2) = $\rho(S_i(m))$，ρ 是一个在 $S_i(m)$ 基调上的改名序偶 (lso_1, lso_2)，调用 ρ 就可以用 (lso_1, lso_2) 来改变 $S_i(m) \cup S_e(m)$ 的基调。其条件为：(i)lso_1 的类子和操作全部来自 $S_i(m)$(即只允许对输入改名)；(ii)ρ 被加进 $S_e(m)\backslash S_i(m)$ 的非继承操作中去，并要改变它们的算术；(iii) 对于 $S_i(m)$ 的每个类子和操作 so，$\rho(so) \in \rho(S_e(m)\backslash S_i(m))$。

这样可以避免新的输入名和非继承输出名之间的混淆，即对某个 so 的改名，不会和非继承类子和操作名混同。输入改名代数系统的语义是：

$$m(\text{m i rename } lso_1 \text{ as } lso_2)(A)\ \rho(so) = m(m)(A|(\rho|S_i(m)))(so)$$

其逻辑含义见图5(假定改名a,b,c为f,g,h)。还可以建立其他结构的复合规约,如e axiom和i axiom是用公理对输入输出语义来进行约束,subset,quotient是造成子集和商集规约。这些结构在组装规约方面的作用不及上面几个重要,为节约篇幅就不加讨论了。

3 复合规约的函数式和逻辑式模型

上面提到,构造说明法的抽象特性来自把某种描述性语言和一个形式机制结合,当然也可以用其它编程语言来描述,但非描述性语言往往涉及有关机器的低级细节,也缺乏直接定义代数类型的手段,程序复杂冗长。因此我们的课题采用函数式语言和逻辑语言来对上述内容进行描述。标准ML语言(SML)是Robin Milner教授等人1985年开发的影响较大的函数型语言,而逻辑语言采用ECC扩展构造演算,即Extended Calculus of Construction,是英国Edinburgh大学计算机系在90年代初提出的一种扩充了的类型理论。显然,这两种模型都非常容易转换为常用编程语言程序。在我们的课题中ML也是系统的元语言。

3.1 复合规约的函数式模型

ML采用一种叫结构(structure)的单元来实现程序模块化。每个结构都有一个定义好的型构(signature),它相当于该结构的类型。我们可以用结构来对应一个抽象数据类型,而用型构来描述它的基调。ML有一个支持动态构造的函子,它是一个从结构到结构的函数,起着粘合结构的作用。我们把它定义为映射输入代数到输出代数的规约操作。

例1 如果有输入结构$I_1,I_2,\cdots,I_n$,函子为EFUN,则输出结构为E可用ML语句表示为:

structure E = EFUN(I_1:I_1SIG and I_2:I_2SIG and…and I_n:I_nSIG)

这一关系完全符合前面的讨论:即

$\rho_1(Alg\Sigma i_1,Alg\Sigma i_2,\cdots,Alg\Sigma i_n)\rightarrow Alg\Sigma e$

下面来看一下用ML实现的复合规约,为节约篇幅仅以合并和组合两种为例,并且输入结构只有两个:

如果spec1和spec2是两个标准规约,其型构分别为spec1SIG和spec2SIG,其输出类子分别为spec1 t_1,spec1 t_2,…,spec1 t_m和spec2 t_1,spec2 t_2,…,spec2 t_n,输出操作为spec1 o_1,spec1 o_2,…,spec1 o_p和spec2 o_1,spec2 o_2,…,spec2 o_q。各个操作所具有的秩分别记作algo spec1 o_1,algo spec1 o_2,…,algo spec1 o_p,algo spec2 o_1,algo spec2 o_2,…,algo spec2 o_q。它们由两个规约中的类子组成,即:

algo spec[1/2] o_i = spec[1/2] t_1 × spec[1/2] t_2 × … × spec[1/2] t_i → spec[1/2] t_{j+1} ($j\geqslant 0$)

例2 ML合并规约的型构为:

```
signature plusSIG =
    sig
      structure spec1:spec1SIG
      structure spec2:spec2SIG
      type spec1 t1
      …
      type spec1 tm
      type spec2 t1
      …
      type spec2 tn
      val spec1 o1:algo spec1 o1
      …
      val spec1 op:algo spec1 op
      val spec2 o1:algo spec1 o1
      …
      val spec2 oq:algo spec1 oq
      end;
```

建立该规约的规约操作可以用ML的Functor来建立:

```
functor plusFUN(structure spec1:spec1SIG and
                structure spec2:spec2SIG)
    sig
      structure spec1 = spec1
      structure spec2 = spec2
      type spec1_t1 = spec1.spec1 t1
      …
      type spec1 tm = spec1.spec1_tm
      type spec2 t1 = spec1.spec1 t1
      …
      type spec2 tn = spec1.spec1 tn
      val spec1_o1 : spec1.spec1_o1
      …
      val spec1 op : spec1.spec1_op
      val spec2 o1 : spec2.spec2_o1
      …
      val spec2_oq : spec2.spec2_oq
      end;
```

例3 ML组合规约的型构为:

```
signature plusSIG =
    sig
      structure spec1:spec1SIG
      structure spec2:spec2SIG
      type spec1 t1
      …
      type spec1 tm
      val spec1 o1:algo spec1 o1
      …
      val spec1 op:algo spec1 op
    end;
```

建立该规约的规约操作的ML的Functor为:

```
functor composeFUN(structure spec1:spec1SIG and
                   structure sepc2:spec2SIG)
    sig
      structure spec1 = spec1 FUN(spec2)
      structure spec2 = spec2.spec2
      type spec1_t1 = spec1.spec1_t1
      …
      type spec1 tm = spec1.spec1 tm
      val spec1 o1 : spec1.spec1 o1
```

```
  ...
  val spec1  op1 : spec1, spec1  op2
end;
```

3.2 复合规约的逻辑式模型

类型理论是进行规约描述的较好工具，它把程序规约、程序构造和程序验证统一于一体。现在除Matin-lof的类型理论外，还有如Eindhoven大学的Automath类型理论，康奈尔大学的Nuprl，以及Coquand-Huet的构造演算等。有的已讨论了程序推导问题，有的也可作为编写规约和程序的语言使用，但是在涉及模块设计和结构化方面，都还没有很好解决，因而类型理论的应有潜力还没有得到很好的开发。

ECC是在Coquand-Huet的构造演算基础上扩充了谓词类型论域和Σ类型(强和类型)，也可认为是在Matin-lof的类型理论上增加了一个表示命题的最低层的非谓词层。这样，ECC实际上就有了与前两者都不同的特性，它包罗了谓词空间和非谓词空间，进一步强化了逻辑公式类型和集合(数据类型)之间的区别，从而形成了一种关于相关类型的统一理论，不但提供了很强的逻辑能力，也提供了很好的抽象机制和结构化机制，如它的类型层次就提供了进行结构化模块化规约设计的手段，把它应用于程序规约，能弥补一般类型理论的不足。现在ECC已有一个很好的开发环境LEGO。

在ECC中，Π类型和Σ类型是两个重要的相关类型。Π类型又叫相关积类型，它提供了相关的函数空间和嵌入逻辑的逻辑公式。另一种相关类型是Σ类型。Σx：A.B(x)，又叫强和类型。直觉上表示序偶(a,b)的集合。在这里，a是类型A的对象，而b是类型B(a)的对象。当B是A上的谓词时，它直觉上就表示类型A的满足特性B的对象的子集[3]。

前面提到，一个基本抽象数据类型的原子规约的基调可以简单地表示为Σ=(S,O)，当我们用ECC来表示一个规约时，可以把(S,O)看成是一个类型序偶。一个标准规约集类型SPEC就写成：

$$SPEC =_{df} \Sigma[Str : Type, Ax : Str \to Prop]$$

在这个类型定义中可见，SPEC集中的一个规约SP是由一个SP的结构类型Str[SP]和在Str[SP]上的谓词Ax[SP]组成的。Str在ECC类型层次中有Type类型，这一类型的对象即是可以实现这个规约的某个可能的结构。而Ax则为一个Π类型。它代表在这类型上的一个谓词，表示该规约在实现时应满足的特性。

对类型理论来说，重要的是对每个规约都要找到它的一个实现，规约SP的一个实现应是某个代数系统s，它也应是类型Str[SP]的一个对象，而这个对象的Ax[SP](s)也是可证明的，SP的结构s和Ax[SP]可满足的证据在一起就叫做SP的一个模型。而SP模型集的类型为：

$$Mod(SP) =_{df} \Sigma s : Str[SP].\ Ax[SP](s)$$

如果模型集非空，则存在一个SP的实现，于是称之为可实现的，或一致的(consistent)。下面我们看一下用ECC的开发环境LEGO来构成复合规约，仍以合并和组合两种为例，并且输入结构只有两个；如果s_1和s_2是两个标准规约，其类型分别为SP和SP'，于是可定义：

```
[STR = Type]1;
[Spec[S : STR] = S- > Prop];
[SPEC =< S : STR > Spec S];
[SP : SPEC];
[Str = SP.1];
[Ax = SP.2];
[Mod =< a : SP.1 > (SP.2 a) : Type];
```

ECC合并规约操作plus可定义为：

```
[Plus [SP,SP':SPEC] =
  (SP.Str # SP'.Str,[s:SP.Str # SP'.Str] and
  (SP.Ax s.1)(SP'Ax s.2):SPEC];
```

可以看出，合并规约由下面两部分组成：

$$Str[plus(SP,SP')] =_{df} Str[SP] \times Str[SP']$$

$$Ax[plus(SP,SP')](s) =_{df} Ax[SP](s.1)\ \&\ Ax[SP](s.2)$$

其中，×即上式中#，是ECC的Σ项，&即上式中and，表示命题的合取规则。规约操作plus本身有类型SPEC → SPEC → SPEC。

ECC组合规约操作compose可定义为：

```
[compose[SP,SP':SPEC] =
  ([s':SP'.Str)]SP.Str  s',  [(s':SP'.Str)]SP.Ax
  s'):SPEC];
```

同样，组合规约也由两部分组成：

$$Str[compose(SP,SP')] =_{df} Str[SP](s')$$

$$Ax[compose(SP,SP')](s) =_{df} Ax[SP](s')$$

这一规约体现了前面分析的语义。这里要注意的是，不要把复合规约compose和通常的compose函数混淆。compose函数是组合两个函数，一般表示为[compose[A,B,C:Type][f:A- > B][g:B- > C] = [x:A]g(fx)]，而组合规约操作compose组合的是两个规约。

(下转封底)