函数程序设计语言用于快速原型

梅 宏 孙永强 (200030上海交通大学计算机系)

摘 要

函数程序设计对减少大型软件系统开发的开销有着较大的潜力。本文讨论了函数程序设计在软件工程中的作用和地位,论述了函数语言作为规范原型工具的合适性,最后用一个例子展示了用函数语言建立系统原型的过

一、引 言

软件危机导致了软件工程的兴起和发展。在传统的软件生存期中,大型软件的开发分成如下几个阶段:1)需求分析,2)规范,3)设计,4)编码,5)测试,6)运行及维护。在这几个阶段中,需求分析和规范是至关重要的两个阶段。需求是用户对未来软件系统的要求,设计人员根据用户需求分析决定系统的设计,因此,一个一经提出就完备不变的需求是十分有用的。然而,由于用户可能对计算机缺乏了解,也可能对自己所需的软件系统的性能缺乏深刻理解,从而可能导致需求随时变化,严重影响整个开发,特别是当系统完成后,用户如果不满意,那么重新按照用户需求返工,将是一项大的耗资工作。

规范是对用户需求的形式描述,是设计人员间精确而有效的通讯手段,也是和最终实现者通讯的工具,因此,对用户需求的规范必须是精确的,也必须易于理解。不精确将导致后期工作的偏差,不易理解则会失去作为通讯媒介的作用。这对形式规范语言提出了较强的要求。

软件生存期方法的瓶颈主要在于这两个阶段。通常为了保证后期工作的不被干扰和最终不返工,设计人员必须和用户反复商讨、分析,以保证需求的完备,一经定下就不允许用户改动。为保证后期工作的准确性,

对需求的形式描述必须力求精确,这也是一项困难的工作。

为了克服生存期方法的缺陷,一种新的软件开发方法,软件原型法应运而生。原型是软件的一个早期可运行版本。原型法采用演示原型的方式来启发、判断、揭示系统的需求。设计人员可在对需求进行简单分析的基础上,对系统快速成型,在原型的演示试用过程中,用户和开发人员间可加强通讯和反馈,通过反复评价和改进原型系统,逐步减少分析和交互中的误解,弥补遗漏,进一步确定各种需求细节,适应需求的变化,最终达到相互间的高度统一,共同建立高质量的系统。

总之,无论是生存期法还是原型法,对需求的规范都是非常重要的,因此,合适的规范工具是必需的;为了构造系统原型,还必须有合适的原型语言。函数程序设计语言是规范语言和原型语言的合适候选。本文将讨论函数语言如何用于形式规范和快速原型,并通过一个例子说明原型过程。

二、关于函数式程序设计语言^[1]

函数语言以Lambda演算为语义基础,其基本机制是函数对参数的作用。函数是程序的基本项,编制程序便是递归构造函数的过程。从数学观点看,函数是从定义域到值域的映射,即函数描述了两个域上元素间的对应关系。因此,函数语言是一种描述性语

言, 它只给出需求解的问题的定义而不需给出具体的求解过程和细节。求解过程是语言本身经过一系列归约规则对程序的应用而实现的。

函数语言具有精确的数学基础, 程序结构简单且表达式具有引用透明性, 由于这些优点, 使得和传统语言相比, 使用函数语言易于编写正确的程序, 程序脱离计算细节, 短小简练, 易理解易维护, 也易于程序演绎和正确性验证。

最早的函数语言是 McCarthy¹1960年创立的Lisp语言, 它对函数语言的发展有着重大影响, Lisp至今仍非常实用和流行。七十年代末开始的函数程序设计研究热潮又导致了很多成功的函数语言的诞生, 如FP、ML、Miranda、Hope、Haskell等。其中Miranda语言是特别典型的, 其方程定义形式的程序尤其易于理解。

如何用函数语言编制大型应用和系统程序的研究一直是一个重要方向, 然而出于目前函数语言效率的考虑, 函数语言编制大型程序总有一定局限。本文旨在说明, 至少在大型软件系统开发中将函数语言作为规范、原型工具是可行的, 且是十分有效的。

三、函数语言用于形式规范

函数程序设计作为软件工程的一个实用方法已有较长历史, 可以追溯到六十年代^[2], 然而, 对于早期计算机的低效率及函数语言的低效实现, 用纯函数方式编写实用程序是不合适的。随着人们对大型程序构造的不断深入理解, 人们意识到抽象数据类型(ADT)等概念已成为大型软件构造的主要概念, 这意味着用纯函数式语言构造大型系统是一个相当自然的选择。在当今, 计算机速度已有了极大提高, 人们关于程序设计的知识也有了很大积累, 以函数程序设计为基础的复杂软件系统的设计应该在软件工程中占有重要地位。当然, 目前就断言函数程序设计能立即应用到软件工程仍是不现实的, 但是, 至少对软件开发中的一个任务——设计

的形式规范, 函数程序设计是有效的。我们主要从以下方面考虑:

1. 软件设计的严密方法要求我们用数学去精确地描述关于系统设计的有关问题。大多数关于形式方法的研究着眼于形式化的程度以便于构造程序正确性的证明。但是数学通常集中考虑计算的算法方面而不是考虑结构。现在流行的软件开发方法如VDM^[3]将对数据和程序结构的考虑放在和算法同等重要的高度, 以满足大型复杂系统设计的需要。我们可以看到, VDM中大型程序形式描述方法的风格类似于递归方程和ADT, 我们完全可以将这个规范看成一个函数程序定义。

2. 作为规范应该忽略细节而着眼于描述对象的行为特性和相互关系, 函数语言显然能满足这个要求, 它只需给出对象的定义, 忽略具体的计算细节。

3. 形式规范的另一作用应是产生一个易于理解的文档以便于设计人员间以及和最终实现者间的有效而精确的通讯, 函数程序的可读性有利于对象的易理解的描述。

4. 规范复用和规范复合也是形式描述的基本要求。它允许对大系统的分解, 在各分量规范的基础上通过复合、复用而形成更大的规范。规范的复合和复用要求规范保持等价, 这符合于函数程序的引用透明性特性。

5. 为支持模块化, 形式描述常采用的机制类似ADT, 而函数语言对ADT的支持是容易的和方便的。

由此, 我们可以看到, 函数语言作为一种形式工具用于规范是合适的。函数语言易于编写正确的程序, 也易于对程序的正确性验证, 这对正确规范的形成是十分有利的。用函数语言形成的规范具有可执行的特征, 我们可以直接将此可执行规范作为原型。

四、可执行规范作为原型

对于一个已建立的规范如何评判其有效性有两种方法, 一是陈述和证明关于规范描述的对象和操作性质的定理; 二是执行规

范。如果我们将函数程序作为规范,则该规范是可执行的。通过将规范作为原型执行,一些不可避免的设计错误和遗漏便可以在设计的早期阶段发现。函数语言用于可执行规范的构造,支持软件设计的快速原型。原型的产生可带来如下好处:

1. 原型可以精化对系统需求的理解,帮助定义用户需求。用户通过和原型的交互,可以能动地投入软件设计活动,在活动中和设计人员一起更好地使需求更趋完备,也更深入地了解系统。

2. 任何仔细的设计都可能留下设计错误和遗漏,通过原型演示可以较早地发现错误和遗漏,及早作出对需求和规范的修正,重建一个新的原型。这是一个重复求精的过程,每次重复都是一次错误的删去和原型的改善,直至最终形成一个完善的令人满意的系统。

3. 原型的执行可建立对系统设计的可信度。

4. 原型可用于对多种设计决策的可行性评价。一个好的设计者应保留探索各种不同定义的机会,从中寻找最精确的定义,只有原型才有可能使设计者有机会去体验不同的设计决策。

函数语言用作规范语言构成可执行规范,进而将可执行规范作为设计的原型,显然,函数程序组合了软件设计的形式规范的明晰性和通过执行使规范生效的能力,是一种较为理想的原型工具。函数语言作为原型语言特别适合于演化式原型方法,通过原型的重复执行达到原型不断进化目的,最终可达到一个令人满意的状态。鉴于目前函数语言已得到较为有效的实现,我们可以将这个最终原型作为我们设计的产品。当然,在效率要求稍高的地方,为达到充分有效的处理机和存贮使用,进一步将设计原型转化为传统语言实现及其他工程工作还是必要的。

五、一个原型构造例子

本节我们借用Miranda语言^[4]的记号作为规范

原型工具来完成一设计工作,建立一个原型。

Ada的程序包是可独立编译的程序分量,它可被开发环境中的编辑器和编译器等访问或修改。每个程序包又分为两个部分,一是接口部分,描述包的对外可视行为,二是实现部分,包括对应接口的具体表示和过程实现。接口和实现均可用imports语句列出其需引用的其他程序包。我们的目的是设计一个管理程序包部件的数据库系统^[5]。

我们原型的设计是给出基本数据对象的规范。

名字表 这是列于imports语句中的程序包名的表。名字间顺序无关,其上有三个基本操作:空表的构成、加入新名、删去名。

```
Namelist = set(Name)
emptylist, → Namelist
addname, Namelist → Name → Namelist
delname, Namelist → Name → Namelist
emptylist = EMPTYLIST
addname x y = x + {y}
delname x y = x - {y}
```

部件 我们将接口和实现用同样的结构表示,表示为一个名、类型和引用名表的三元组,其中类型决定其为接口部件还是实现部件。部件上的操作有部件构造和部件分量提取。

```
Part = (Name, Kind, Namelist)
mkpart, Name → Kind → Namelist → Part
nameof, Part → Name
kindof, Part → Kind
namelistof, Part → Namelist
mkpart x y z = (x, y, z)
nameof (x, y, z) = x
kindof (x, y, z) = y
namelistof (x, y, z) = z
```

数据库 数据库中存贮的内容为部件,部件间顺序无关。库上的基本操作有:空库的构造、加入新部件、删去指定部件、取某接口部件、取某实现部件、查询某程序包接口部件是否存在,查询某程序包实现部件是否存在、判定某程序包能否被编译(编译的前提是其实现和引用程序包的实现的传递闭包均存在)等。

```
DB = set(Part)
emptydb, → DB
addpart, DB → Part → DB
delpart, DB → Part → DB
getinterface, DB → Name → Part
```

```

getimpln, DB→Name→Part
existinterface, DB→Name→Bool
existimpln, DB→Name→Bool
compilepart, DB→Name→Bool
emptydb=EMPTYDB
addpart x y=x+{y}
delpart x y=x-{y}
getinterface (addpart x p)n=p,
  nameof p=n ∧ kindof p=interface
  =getinterface x n, otherwise
getimpln(addpart x p)n=p,
  nameof p=n ∧ kind of p=impln
  =getimpln x n, otherwise
existinterface EMPTYDB n=F
existinterface (addpart x p)n=T,
  nameof p=n ∧ kindof p=interface
  =existinterface x n, otherwise
existimpln EMPTYDB n.=F
existimpln (addpart xp) n=T,
  nameof p=n∧kindof p=impln
  =existimpln x n, otherwise
compilepart xn=F, not existimpln xn
  =and{compilepart x n' |n'}
  ←namelist of{getimpln x n}, otherwise

```

此处and是对集合上所有元素的操作。

至此,我们已经通过对需求的分析建立了一个规范,这个可执行规范可视为一个原型,然而,在我们的设计中,还存在一些遗漏和错误,如addpart有处理同名冲突的能力,delpart的参数要求太严,getinterface和getimpln的定义没有考虑空库情形等,这些错误均可通过原型的执行被检测出来,一旦发现这些错误,则我们可以通过对规范的修改,使原型进化为一个新的原型,这将是一次质的改善。而在新原型的基础上,新一轮的检测演示又可继续进行。

六、结束语

本文讨论了函数语言作为规范、原型语

言的潜在优势及函数程序设计在软件工程中的应用。函数程序结合了形式规范的明晰性要求和通过执行判定规范有效性的能力。在需求分析基础上,函数语言直接建立可执行规范作为系统原型,是一种合适的快速原型工具,这对大型软件系统设计是十分有用的。

参考文献

- [1] 梅宏,孙永强,函数式、逻辑式和对象式程序设计及其多范例合成语言,计算机科学,1991.第五期
- [2] J.McCarthy, Recursive functions of symbolic expression and their computation by machine,commun. ACM, Vol. 3, No.4, 1960
- [3] D.Bjorner et al., Formal specification and software development, Englewood Cliffs, NJ, Prentice-Hall, 1982
- [4] D.Turner, Miranda, A non-strict functional language with polymorphic types, LNCS 201, New York, Springer-Verlag, 1985
- [5] R.M.Gallimore et al., UMIST OBJ, a language for executable program specifications, The Computer Journal, Vol. 32, No.5, 1989
- [6] P.Henderson, Functional Programming, Formal Specification, and Rapid Prototyping, IEEE Transactions on S.E. Vol. SE-12, No.2, 1986
- [7] V.Breazu-Tannen et al., Typed Functional Programming for Rapid Development of Reliable Software, 1988, 27th Symp. of the Washington, D.C. Chapter of the ACM and National Bureau of Standards