使递归算法泛型化

缪伟宇,邵志清
(华东理工大学 信息科学与工程学院,上海 200237)

摘 要:对于泛型程序设计来说,类型理论中的参数化多态是其理论框架,因为参数化多态引入了类型变量,使得类型参数化,从而完全支持类型上的抽象。然而对于现行的泛型算法,无论是C++标准模版库中的泛型算法还是基于函数式程序设计语言的算法,函数功能的定义比较具体化、单一化,因而缺乏可扩展性和高度的复用性。将对递归算法进行抽象,构造原始递归构造子,使得一般的泛型算法都可以通过该算子来构造,从而加强泛型算法的可复用型与可扩展性。除此之外,分析了递归算法构造子与泛型程序设计中的iterator概念和用于描叙泛型概念的形式化语言Tecton中所提倡的reuse概念的一致性。也给出算法复杂度的定量分析,并用函数式语言ML来实现。

关键词:泛型编程;泛型算法;原始递归;函数式程序设计
中图分类号:TP301.6 **文献标识码**:A **文章编号**:1673-629X(2008)07-0096-04

Making Recursive Algorithms Generic

MIAO Wei-yu, SHAO Zhi-qing
(Sch. of Info. Sci. and Eng., East China Univ. of Sci. and Tech., Shanghai 200237, China)

Abstract: Parametric polymorphism is crucial to generic programming in that it adopts parameterized types and therefore fully supports the abstraction on types. However, current generic algorithms, both developed in C++ standard template library and in functional programming, have sole and concrete functional definitions, which lack scalability and reusability. Presents the abstraction on algorithms and primitive recursion constructor, aiming to reinforce the high reusability and extensibility of generic algorithms. In addition, analyzes the consistency of this algorithmic abstraction with generic iterator concepts and Tecton concepts, presents quantitative time-complexity analysis, and gives the practical implementation in ML.

Key words: generic programming; generic algorithms; primitive recursion; functional programming

0 引 言

参数化多态[1,2]于20世纪引入类型理论。参数化多态对于泛型编程来说至关重要,因为它使得类型参数化,从而支持类型上的抽象:用类型变量(模版参数)来替代那些实例化的类型,当使用时,这些类型变量由编译器重新实例化为指定的实际类型。首先,介绍参数化多态的理论框架,即类型系统 F 的语法:

类型 $T,U ::= X \mid \tau \mid T * U \mid T \rightarrow U \mid \mu X.T \mid \forall X.T$

原子类型 $\tau ::= \mathrm{Bool} \mid \mathrm{Real} \mid \mathrm{Int} \mid 1$

项 $t,u := x \mid v \mid < t,u > \mid \lambda x.t \mid tu \mid \Lambda X.t \mid t < X >$

变量上下文 $\Gamma ::= \cdot \mid \Gamma, x : T \mid \Gamma, X$

类型规则 $\Gamma, x \mid - x : T$

类型包括类型变量、原子类型、序对类型、函数类型、递归类型和全称类型。原子类型包括布尔类型、实数类型、整数类型和单位unit类型。项包括项变量、值、序对、项的lambda抽象、项的lambda应用、类型的lambda抽象和类型的lambda应用。变量上下文包括类型变量和项变量的序列。类型规则包括项的lambda抽象和应用规则:$\Gamma \mid - \lambda x.t : T_1 \rightarrow T_2, \Gamma \mid - u : T_1, (\lambda x.t)u \equiv [x \mapsto u]t$ 以及类型的lambda抽象和应用规则:$\Gamma \mid - \Lambda X.t : \forall X.T_1, (\Lambda X.t) < T_2 > \equiv [X \rightarrow T_2]t$。

参数化多态使得算法和数据结构不依赖于特定的类型,因此泛型算法可以作用于不同数据类型的数据结构且不用重写代码。然而,现在的泛型算法在功能的定义上比较单一、具体和独立,因此导致这些算法抽象程度底、可扩展性差。为了解决上述的问题,引入了高阶抽象和结构化多态。尽管以前的学术论文[3,4]有

收稿日期:2007-10-12
基金项目:国家自然科学基金资助项目(60373075)
作者简介:缪伟宇(1983-),男,硕士研究生,研究领域为泛型编程和程序设计语言;邵志清,博士,教授,研究领域为软件开发与验证方法。

这方面的涉及,但是只是涉及一些特例,没有理论的分析,也没有泛型概念的叙述。文中将通过以下几方面来弥补以前的不足:分析高阶抽象与原始递归的联系,为算法抽象的机制构建理论框架;用形式化语言 Tecton 来描述算法抽象并用 iterator 概念来分析原始递归构造算子 unfold,从而表明文中的算法抽象与泛型概念的一致性;在应用方面,用函数式语言 ML 实现并进行复杂度分析。

1　基于原始递归的算法抽象

为了了解使递归算法泛型化的必要性,首先给出一些例子。通过这些例子,可以发现这些递归算法结构的相似性。仅在下面的例子中,将类型变量显式地表示出来以突出参数化多态(类型变量)机制的引入;而在以后的章节中,我们的描述将近似于 ML 语言,将不把类型变量显式地表示。

例1　递归函数 AppendElem 将一个元素添加到一个泛型链表的尾部。如果函数接受一个空的链表,则直接用链表构造子 Cons 将元素插入空链表;否则用 Cons 函数,通过递归重新构造新链表。

```
AppendElem<X> Elem Nil = Cons<X> Elem Nil
AppendElem<X> Elem (Cons<X> x List<X>) = Cons<X> x (AppendElem<X> Elem List<X>)
> Cons<X> : ∀X.X→(List X)→(List X)
> AppendElem<X> : ∀X.X→(List X)→(List X)
```

例2　递归函数 ReverseList 通过上述定义的 AppendElem 函数将链表中元素的排列顺序颠倒。

```
ReverseList<X> Nil = Nil
ReverseList<X> (Cons<X> x List<X>) = AppendElem<X> x (ReverseList<X> List<X>)
> AppendElem<X> : ∀X.X→(List X)→(List X)
> ReverseList<X> : ∀X.(List X)→(List X)
```

例3　递归函数 InsertionSortList 按照插入排序算法,通过 InsertionInOrder 函数(将一个元素有序地插入一个已排序好的链表中)排序一个链表。

```
InsertionSortList<X> Nil = Nil
InsertionSortList<X> (Cons<X> x List<X>) = InsertionInOrder<X> x (InsertionSortList<X> List<X>)
> InsertionInOrder<X> : ∀X.X→(List X)→(List X)
> InsertionSortList<X> : ∀X.(List X)→(List X)
```

从以上例子可以得出所有的原始递归形式有着相同的特征:所有的原始递归由基函数(Base)和生成构造子(Generating Operator, GOper)构成。假设 h 是通用的原始递归函数,f 为基函数,g 为生成构造子,则:

$$h(x,y_1) = f(x),\ h(x,y_n) = g(x,y_n,h(x,y_{(n-1)}))$$

$$h = \mathrm{rec}(f,g)$$

当进一步观察上面给出的三个例子,不难发现例子中的递归构造与 x 无关且 g 是二元函数。虽然表达能力上较通用的原始递归函数有所减弱(如无法表达 Ackmen 函数),但是足以表达大部分的递归算法。接下来,给出该递归函数的构造子 unfold,unfold 作用于一个原子类型的项(Base),一个二元函数类型的项(GOper)和一个递归类型($\mu X.T$)的容器。

```
Base: T_Base
List: (T_List = μR. ∀X.(X * R))
GOper: (T_GOper = ∀X. ∀Y.X→Y→Y)
Unfold: T_GOper→T_Base→T_List→T_Base
Unfold≡Unfold GOper Base Nil = Base
  Unfold GOper Base (Cons x List) = GOper x(Unfold GOper Base List)
Unfold≡λGOper.λBase.λList.
    match List with Nil⇒Base;
    | Cons(x, List')⇒GOper(x, Unfold(GOper, Base, List'));
```

unfold 概念原先应用于递归类型[1,5,6]:对某个递归类型 $\mu X.T$ 的展开,即 unfold 操作就是将结构体 T 中所有的 X 用递归类型本身来替代。比如 unfold($\mu X.T$) = $[X \mapsto \mu X.T]T$,其中 $\mapsto$ 是标准的替代符号。这里将 unfold 概念移植到含递归结构的容器上,从而可以遍历容器中的所有元素。以下是展开一个泛型链表的操作:

$$\begin{aligned}\mathrm{unfold}(\mathrm{List}_1) &= \mathrm{Cons}\ e_1\ \mathrm{unfold}\ (\mathrm{List}_2)\\ &= \mathrm{Cons}\ e_1\ \mathrm{Cons}\ e_2\ \mathrm{unfold}\ (\mathrm{List}_3)\\ &= \mathrm{Cons}\ e_1\ \mathrm{Cons}\ e_2\ \mathrm{Cons}\ e_3\ \mathrm{unfold}\ (\mathrm{List}_4)\\ &\vdots\\ &= \mathrm{Cons}\ e_1\ \mathrm{Cons}\ e_2\ \mathrm{Cons}\ e_3 \ldots \mathrm{Cons}\ e_{(n-1)}\ \mathrm{unfold}\ (\mathrm{List}_n)\\ &= \mathrm{Cons}\ e_1\ \mathrm{Cons}\ e_2\ \mathrm{Cons}\ e_3 \ldots \mathrm{Cons}\ e_{(n-1)}\ \mathrm{Nil}\end{aligned}$$

除了本节开始所介绍的算法之外,可以用 unfold 递归算法构造子构造许递归算法,例如 SumList 对泛型链表中所有的元素求和;MapList 映射泛型链表中的每个元素;ConcatenateList 将一个泛型链表添加到另一个泛型链表之后;FilterList 将链表中不符合条件的元素去除。

```
AppendElem≡λElem.λList.(unfold Cons (Cons Elem Nil) List)
Reverse≡λListList.(unfold AppendElem Nil List)
InsertionSortList≡λList.(unfold InsertionInOrder Nil List)
SumList≡λIdentity.λList.(unfold Plus Identity List)
MapList≡λUnaryFunction.λList.(unfold (Compose1st Cons UnaryFunction) Nil List)
ConcatenateList≡λList.λList'.(unfold Cons List' List)
FilterList≡λUnaryPredicate.λList.(unfold (Binder3rd ConditionalCons UnaryPredicate) Nil List)
```

2 支持 Tecton 的使用概念

泛型编程是一种开发通用的、高适应性的、和高复用性的软件构件技术,因此可复用性是泛型概念的核心。这点在 Tecton 语言[7,8]的使用概念(use concept)上明显地反映出来。泛型概念不提倡 Tecton 语言中的优化概念(refinement concept),因为优化有着与类的性质相似的继承和层次结构关系,从而容易导致优化过程是建立在脆弱或错误的基类之上。文中的算法抽象对 Tecton 的使用概念充分的支持,因为该抽象运用操作之间的纯粹的组合,而避免了操作之间的层次关系。用 Tecton 语言描述了通过递归抽象所构造的算法,以下是核心部分。

<AppendElem, Using List, Unfold>≡

Abbreviation : AppendElem is Unfold [with Cons as GOper, Cons(Elem, Nil) as Base, List as Sequence];

<ReverseList, Using List, AppendElem, Unfold>≡

Abbreviation : ReverseList is Unfold [with AppendElem as GOper, Nil as Base, List as Sequence];

<SumList, Using Identity, Plus, list, Unfold>≡

Abbreviation : SumList is Unfold [with Plus as GOper, Identity as Base, List as Sequence];

<MapList, Using List, Unary - op, Compose1st, Unfold>≡

Abbreviation : MapList is Unfold [with Compose1st (Cons Unary - op) as GOper, Nil as Base, List as Sequence];

<ConcatenateList, Using List, Unfold>≡

Abbreviation : ConcatenateList is Unfold [with Cons as GOper, List' as Base, List as Sequence];

<FilterList, Using Binder3rd, Conditional - cons, Unary - predicate, List, Unfold>≡

Abbreviation : FilterList is Unfold [with Binder3rd(Conditional - cons, Unary - predicate) as GOper, Nil as Base, List as Sequence];

<InsertionSortList, Using Insertion - in - order, List, Unfold>≡

Abbreviation : InsertionSortList is Unfold [with Insertion - in - order as GOper, Nil as Base, List as Sequence];

根据 Tecton 语言中的使用概念,不难发现许多 STL 中的具体算法都可以用操作的组合来实现:Unfold 函数,Plus 函数和返回常值为 1 的函数可以组成 Count 函数。所有的 Find,Find if,Remove 和 Remove if 可以由 FilterList 函数来构建。Count if 可以由 FilterList 和 Count 来构建。For each 可以由 MapList 来构建。还可以构建 Replace,Rotate 等等。

3 支持 Iterator 概念

从本质上来看,unfold 遍历了泛型链表的每个元素。这与 iterator 的概念一致:iterator 是容器与算法的接口,算法通过 iterator 对容器的元素进行访问和操作。iterator 使数据结构与算法互相独立,且算法更为通用。已经定义了泛型链表的 unfold 函数。除了泛型链表之外,可以定义其它一维数据结构的 unfold 函数,如 vector,stack,set 等。也可以建立二维数据结构的 unfold 函数,如二叉树:

```
Unfold bintree≡λGOper. λBase. λBintree.
    match Bintree with leaf⇒Base;
    | node(x, leftsub, rightsub) ⇒ GOper(x, Unfold bintree (GOper, Base, leftsub), Unfold bintree(GOper, Base, rightsub));
```

则可以利用 Unfold bintree 去定义一些针对树结构的操作,如 Reverse bintree 函数对每个节点的左右分支进行交换,Map bintree 函数将一颗二叉树映射到另一棵二叉树,Traverse bintree 函数将二叉树结构转换成链表(tree node traverse),等等。

Reverse bintree≡λBintree. (Unfold bintree ReverseNode leaf Bintree) 其中 ReverseNode≡λValue. λLeft. λRight. (node Value Right Left)

Map bintree ≡ λUnaryFunction. λBintree. (Unfold bintree (Compose1st Node UnaryFunction) leaf Bintree)

Traverse bintree left - mid - right≡λBintree. (Unfold bintree (LMR - traverse) Nil Bintree) 其中 LMR - traverse≡λValue. λLeft. λRight. (ConcatenateList Left (ConcatenateList (Cons Value Nil) Right))

Traverse bintree left - right - mid≡λBintree. (Unfold bintree (LRM - traverse) Nil Bintree) 其中 LRM - traverse≡λValue. λLeft. λRight. (ConcatenateList (ConcatenateList Left Right) (Cons Value Nil))

Traverse bintree mid - left - right≡λBintree. (Unfold bintree (MLR - traverse) Nil Bintree) 其中 MLR - traverse≡λValue. λLeft. λRight. (ConcatenateList (Cons Value Nil) (ConcatenateList Left Right))

因此,unfold 递归算法可以归结为两类:一类是对一维数据结构的操作,另一类是对二维数据结构的操作。

4 通过 ML 语言实现和时间复杂度分析

选择用 ML 语言来实现递归算法的抽象,理由如下:

首先,ML 是高阶函数式语言。ML 既支持内嵌式函数(nested function),内部函数可以调用外部函数的变量;也支持函数作为返回值。这两个特征的组合定义了高阶函数的概念。因此,可以实现 currying 而不需要函数绑定机制。其次,ML 和 C++,Generic Java 一样支持抽象数据类型和参数化多态。此外,ML 的类型定义精确且简洁。最后,ML 在模块化程序设计

表1 尾递归形式与尾递归形式的时间复杂度比较(μs)

Elements Num.	ReverseList		MapList		FilterList		InsertionSortList		SumList	
	NonTail	Tail	NonTail	Tail	NonTail	Tail	NonTail	Tail	NonTail	Tail
10	17825	18025	17575	15825	2400	2655	17775	17925	755	750
50	18275	18125	19525	17625	20080	20230	18575	18680	800	800
100	19530	18180	21335	19130	20130	20330	20680	20830	850	805
500	65280	18620	22020	19820	22440	22420	96340	95940	1400	1200
1000	246400	19000	23100	21000	24000	24100	373600	368500	2000	2000
5000	11371500	25000	30000	30500	35000	30000	14946500	14090500	10000	5000

(modular programming)[9]方面相当的成熟。因此支持函数的组合来构造新的函数。

unfold递归算法的时间复杂度是二次方(n^2)。虽然将unfold算法改为尾递归(tail recursion)形式从理论上可以把时间复杂度降低到等同于循环结构,但是需要添加额外的工作:由于尾递归形式使得一维数据结构存储元素的顺序颠倒,需要额外地调用ReverseList函数来纠正顺序。

```
Unfold - tailrecUnfold - tailrec GOper Base Nil = Base
    Unfold - tailrec GOper Base (Cons x List) = Unfold - tailrec
GOper (GOper x Base) List
```

表1给出了非尾递归形式与尾递归形式的时间复杂度的实验比较数据(Pentium 4 1.8GHz平台)。

从数据来看,除了ReverseList之外,尾递归对于其他的算法在时间复杂度上优势并不明显。此外,递归抽象算法随着元素增加,时间复杂度增长温和,具有可行性。

5 结束语

递归算法构造子unfold提升了泛型算法的抽象程度。该抽象不但可以提高软件构建的复用度,而且也可以用于构造泛型递归算法,扩充算法库且不需要重复构造结构、功能相似的算法。此外,在带证明的代码系统(proof - carrying code)的研究领域,递归算法的抽象可以用于构造复用性和抽象度比较高的公理和证明,从而可以减少构造公理和算法的繁琐过程。

将来的工作也可以包括对生成算子的研究和利用该抽象用于证明算法结构的相等性,从而证明算法功能的等价性。

参考文献:

[1] Pierce B. Types and programming languages[M]. [s. l.]: MIT Press, 2002.

[2] Pitts A. Parametric polymorphism and operational equivalence [J]. Mathematical Structures in Computer Science, 2000 (10):231 - 259.

[3] Gibbons J. Design patterns as higher - order datatype - generic programs[C]// International Conference on Functional Programming, Proceedings of 2006 ACM SIGPLAN Workshop on Generic Programming. Portland, Oregon, USA:[s. n.], 2006:1 - 12.

[4] Ruehr K. Analytical and structural polymorphism expressed using patterns over types[D]. USA:University of Michigan, 1992.

[5] Pierce B. Advanced topics in types and programming languages[M]. [s. l.]: MIT Press, 2005.

[6] Sumii E, Pierce B. A bisimulation for type abstraction and recursion[J]. Journal of the ACM,2007,54(5):26 - 30.

[7] Musser D, Shao Zhiqing. Concept use or concept refinement: an important distinction in building generic specifications[C] //4th International Conference on Formal Engineering Methods, volume 2495 of Lecture Notes in Computer Science. New York: Springer - Verlag, 2002.

[8] Musser D, Shao Zhiqing. The Tecton concept description language (revised version)[R]. Computer Science Department, Rensselaer Polytechnic Institute. Troy, New York:[s. n.], 2002.

[9] Dreyer D, Crary K, Harper R. A type system for higher - order modules[C]//30th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Langauges (POPL'03). New Orleans, Louisiana, USA:[s. n.],2003:236 - 249.

(上接第95页)

Mining: Discovering Process Models from Event Logs[J]. IEEE Transactions on Knowledge and Data Engineering, 2004,16(9):1128 - 1142.

[3] Herbst J, Karagiannis D. Workflow mining with InWoLve [J]. Computers in Industry,2004,53(3):245 - 264.

[4] Schimm G. Mining exact models of concurrent workflows[J]. Computers in Industry,2004,53(3):265 - 281.

[5] Hammori M, Herbst J, Kleiner N. Interactive workflow mining - requirements, concepts and implementation[J]. Data and Knowledge Engineering, 2006,56(1):41 - 63.