计算机科学 1996Vol 23 №.6

多方法分派研究现状*)

梅 宏 黄小晗 杨芙清

(北京大学计算机科学技术系 北京 100871)

摘 要 In the research on Object-Oriented Programming Language, multi-methods has become an important aspect in which method-dispatching is essential and difficult. This paper presents an overview of current research on the method-dispatching of multi-methods. The discussion focus on three kinds of relatively effective algorithms.

关键词 Object-Oriented Programming Language, Multi-Methods, Method-Dispatching, Subtype.

一、引 言

近十年来,面向对象技术得到了迅速的发展,目前已经实现的面向对象(OO)程序设计语言数以百计,根据不同的视角可对其分类。例如基于类的OO语言(如Smalltalk objective-c等)、基于原型的OO语言(如SELF等)、混合型OO语言(如C++等)和纯OO语言(如Smalltalk等)。随着网络技术的发展,又出现了分布式OO语言(如Amer等)。近年来,人们根据接收消息的对象是否单一又把OO语言划分为基于单方法(mono-method)的OO语言和基于多方法(multi-methods)的OO语言。像C++、Smalltalk等非常流行的OO语言是基于单方法的;而基于多方法的OO语言是在最近十年中迅速发展起来的,其中最典型的代表是CLOS,其他著名的还有Cecil、SELF、Flavor及CLOS的前身Commonloops等。目前,支持多方法的OO语言正逐步成为OO语言中的一个研究热点。标准数据库查询语言SQL3的提议中,已把多方法作为其中的一部分。

多方法[1]是对单方法(或selfish method)的扩充。在多方法环境下,消息被传递给所有的参数,而单方法情况下,只有一个接收消息的对象(receiver或target)。因此,在多方法环境下,一个类属函数(generic function)的具体执行是由所有的参数共同来决定的。

多方法的引入增强了OO语言的功能,使其具有更强的表达能力,在许多情况下能更客观地反映客观事物之间的相互关系。例如,假设我们定义了两个类School和Enterprise,现在有一个消息Contract(其任务是在School和Enterprise之间签订一个合同)。那么,这个消息应该传给哪一个类(或对象)呢?换句话说,方法Contract应该封装在哪一个类中呢?显而易见,最直观的方法是把School和Enterprise都作为消息的接收者,即采用多方法。当然,一些基于单方法的OO语言提供了某些特性来解决类似的问题。比如C++中提供了友元函数。然而,如果把Contract定义为类School和Enterprise的友元函数,与采用多方法相比较,至少有两方面的不足:1)不能准确直接地表达现实事物之间的关系;2)不具有继承的特性。

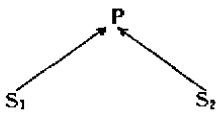
多方法保留了全部继承特性,使得基于多方法的OO语言具有更强的多态性,便于程序员写出代码量小而又便于理解的程序。另外,强的多态性对于领域应用框架和体系结构的开发是非常有用的支持。例如类属函数display(shape,device),在执行过程中可根据shape(可能是circle,triangle,rectangle……)和device(可能是CRT,Window,Printer……)的实际情况选择相应的方法。

当然,基于多方法的语言也存在着一些不足之处[2],比如语义复杂,与OO风格及封装性相悖[3],效率偏低等。其中效率低下是导致目前多方法OO语言尚未广泛接受和使用的最根本的制约因素,究其效率低的主要原因就是方法分派。

*)本文受863高技术计划及教委博士点基金资助。

二、方法分派

动态绑定(binding)是OO语言的核心特征之一，其实现依赖于方法分派机制。所谓方法分派，就是在运行时刻对某个类属函数调用选择一个最具体可运行(MSA—most specific applicable)方法。从消息传递的机制来看，也就是当一个消息传递给一个或多个接收消息的对象时，选择最恰当的响应方法。下面我们举例说明方法分派的基本过程。



单方法：$m_1(P)$
$m_2(S_2)$
多方法：$m_1(P,P)$
$m_2(S_1,S_2)$
$m_3(S_1,P)$

图1

首先给出一些基本概念[16]：

·子类型关系。用符号"≤"表示子类型关系(一种偏序关系)。A≤B表示A是B的子类型或者说B是A的超类型(supertype)。

·方法可应用性。方法 $m_k(T_1^k,\cdots,T_n^k)$ 可应用于类属函数调用 $m(T^1,\cdots,T^n)$，当且仅当 $\forall i, 1\leqslant i\leqslant n$，$T^i\leqslant T_i^k$。(*注意：本文中方法名的下标都是为区别起见而额外加上去的)

·方法具体性。如果有多个方法可应用于某个类属函数调用，我们必须确定它们之间的优先顺序。假设方法 $m_1(T_1^1,T_1^2,\cdots,T_1^n)$ 和方法 $m_2(T_2^1,T_2^2,\cdots,T_2^n)$ 都可应用于某个类属函数调用，并且满足 $\forall i, 1\leqslant i\leqslant n, T_1^i\leqslant T_2^i$，则称 m_1 比 m_2 更具体(more specific)。然而，仅仅依靠子类型关系是不足以完全确定方法的具体性的。R. Agrawal 等人概括了用于确定方法具体性的六种方法，在此不再赘述。

有了以上的基本概念，我们就可以来介绍方法分派的过程了。图1给出了一个非常简单的类型结构图及其对应的方法。对单方法调用 m(T)(可能还带有其他不影响分派的参数)，在运行时可能有表1所示几种情况；多方法调用 $m(T^1,T^2)$ 的情况见表2。

表1

T的类型	可应用的方法	MSA
P	m_1	m_1
S_1	m_1	m_1
S_2	m_1,m_2	m_2

表2

T^1 的类型	T^2 的类型	可应用的方法	MSA
P	P	m_1	m_1
P	S_1	m_1	m_1
P	S_2	m_1	m_1
S_1	P	m_1,m_3	m_3
S_1	S_1	m_1	m_1
S_1	S_2	m_1,m_2,m_3	m_2
S_2	P	m_1	m_1
S_2	S_1	m_1	m_1
S_2	S_2	m_1	m_1

运行时的动态方法分派支持语言的多态性，但分派的耗费很大，降低了可执行目标代码的时间和空间性能，这个问题在多方法语言中尤为突出。因此，寻找高效的分派算法就成了研究方法分派的首要目标。

选择方法分派算法要对时间和空间性能进行权衡，对于不同的侧重点，有两类不同的技术[4]：

1)非常量时间技术。一般说来，非常量时间技术不需要太多的额外运行空间，但其动态查找所需的时间较长，且查找时间依不同的情形而变化。

模式搜索是一种简单的处理方法。对单方法调用 m(T,…)，分派时首先查找T类的方法集合，若未找到则查找其父类(在多继承的情形下，依据某种规则确定查找顺序)。这个过程持续到已查到该方法或不再有父类为止。对多方法调用则不能采用上述过程。多方法模式搜索过程是这样的：找出该类属函数所有的方法，并进行排序，从最具体到最不具体。第一个可应用的方法就是MSA方法，无论单方法还是多方法，模式搜索都要耗费大量的运行时间。

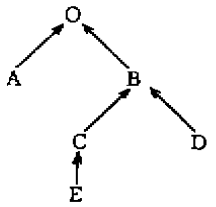
为了避免不必要的搜索，节省方法分派的平均时间，研究者提出了一些优化技术，其中最重要的是缓冲(Caching)和内嵌(Inlining)技术。前者是把先前使用过的方法存入缓冲区中，在搜索时首先查找缓冲区，若找不到再根据相应的搜索机制进一步查找。人们已提出好几种缓冲技术，应用于 Smalltalk 和 Sather 的单一全局缓冲[5][6][7]；在 CLOS 和 Sather 中使用的局部缓冲(局部于一个类属函数或一个调用)[7][8]；Smalltalk 和 Self 使用的在线缓冲[9][10][11]等。一个系统通常把这几种技术加以组合使用。内嵌技术是指通过类型分析和运行时类型检查以尽量避免方法分派。现已提出几种相当复杂的技术来实现内嵌：类型预测[6][9]、Smalltalk 中的类型包装(type casing)[12]以及用于 SELF 中的定制(customization)

和分割(splitting)[10][13][14][15]。

非常量时间技术采用的是"以时间换取空间"的策略,一般难于应用于实时系统。

2)常量时间技术——分派表。与非常量时间技术相反,常量时间技术更加注重于目标代码的时间性能。最常用的常量时间技术是分派表。

①单方法分派表。图2所示类结构的方法分派表如图3所示。在程序运行时,用类属函数名及对应参数的类型作为索引查找分派表,就可以找到对应的MSA方法。



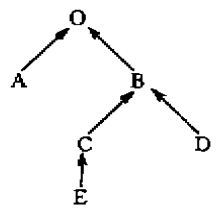
o_1(O) o_2(A) o_3(B)
a_1(A)
b_1(B) b_2(D)
c_1(C)

图2

方法 \ 参数	O	A	B	C	D	E
o	o_1	o_2	o_3	o_3	o_3	o_3
a	—	a_1	—	—	—	—
b	—	—	b_1	b_1	b_2	b_1
c	—	—	—	c_1	—	c_1

图3

②多方法分派表。因为多方法分派表涉及到类属函数的多个参数,且参数个数不等,因此必须为每个类属函数保存一张分派表。一个n元(arity)类属函数的分派表是一个n维数组。图4所示类结构所对应的二元类属函数m的分派表如下(以下都用整数i代替方法m_i)。



m_1(A,B)
m_2(B,C)
m_3(C,D)

图4

	O	A	B	C	D	E
O	—	—	—	—	—	—
A	—	—	1	1	1	1
B	—	—	—	2	2	2
C	—	—	—	2	3	3
D	—	—	—	2	2	2
E	—	—	—	2	3	3

图5

在程序运行时,首先根据类属函数名及其元数找到对应的分派表,然后以参数类型名为索引查找分派表就可以找到对应的MSA方法。

采用分派表可以很快地完成动态分派,但是分派表占用了大量的运行空间(单分派时所占空间为:类属函数个数×类型个数;多分派时每个n元类属函数所占空间为类型个数的n次方),多分派时则直接导致了组合爆炸。因此,有必要对分派表进行压缩。

着色法(coloring)[17]比较成功地对单方法分派表进行了压缩。它的基本思想是删除分配表中的大量空表项。算法的一个不足之处是耗费了大量的编译时间[18]。

近几年来,人们在多方法分派算法的研究中开展了许多有益的工作,在下一部分,我们将介绍其中有代表性的三种算法思想。

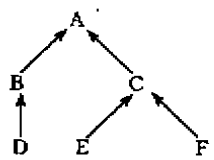
三、多方法分派的近期主要研究进展

方法分派是多方法研究中的重点和难点,目前人们已经在这个领域取得了一些进展,但仍有大量的工作需要人们进一步去努力。在这一部分,我们将介绍三项具有代表性的工作,并对它们的优缺点进行分析探讨。

1 静态类型检查法

这一小节介绍R. Agrawal, L. G. Demichiel和B. G. Lindsay的工作[16]。

1.1 算法思想 静态类型检查法的基本思想是在编译过程中,通过类型检查,确定一个可应用的方法子集,动态分派时只需在此方法子集中查找MSA方法。可以用一个简单的例子来说明这种思想。



m_1(A,A)
m_2(B,B)
m_3(C,C)
m_4(D,D)
m_5(E,E)
m_6(F,F)

图6

上图所示类结构所对应的类属函数m共包含六个方法,我们给出它的一个有效的拓扑排序(即满足条件:排在后面的方法不比排在前面的方法更具体)$\{m_6, m_5, m_4, m_3, m_2, m_1\}$。对于函数调用m(b,b),我们只需保存这个拓扑排序的子集$\{m_4, m_2\}$,在执行时根据参数的实际类型确定MSA方法。

1.2 **算法基本步骤** 首先我们介绍几个基本概念及定理。

· 混淆性(confusability) 一个类属函数 m 的两个方法 $m_1(T_1^1,\cdots,T_1^n)$ 与 $m_2(T_2^1,\cdots,T_2^n)$ 是混淆的，当且仅当 $\forall i,1\leqslant i\leqslant n$，存在类型 T^i，$T^i\leqslant T_1^i \wedge T^i\leqslant T_2^i$。

· 混淆集(confusable set) 一个类属函数 m 的一个混淆集是满足以下条件的最大集合 C：如果 m_i，$m_j\in C$，则 $\exists\ m_1,\cdots,m_k(k\geqslant 0)$，$m_i$ 与 m_1 是混淆的，m_1 与 m_2 是混淆的……，m_k 与 m_j 是混淆的。

· 覆盖(cover) 一个类属函数调用 m 被一个混淆集 C 所覆盖，当且仅当 $\exists\ m_s\in C$，m_s 可应用于 m。关于混淆集有一条重要的定理：

定理 1 一个类属函数调用只能被最多一个混淆集所覆盖。

· 顶集(tops) 给定一个 n 元类属函数 m 的一个混淆集 S，对 $1\leqslant i\leqslant n$，tops(S,i) 定义为集合 $\{T\mid \exists\ m_j,(T_j^1\cdots\cdots,T_j^n)\in S,T=T_j^i\wedge\exists\ m_k(T_k^1,\cdots\cdots T_k^k)\in S,T<T_k^i\}$。关于顶集也有一个重要的定理：

定理 2 如果一个类属函数调用 $m(T^1,\cdots,T^n)$ 被一个混淆集 S 所覆盖，则 $\forall i,1\leqslant i\leqslant n,\exists\ T\mid T\in tops(s,i)\wedge T^i\leqslant T$。

基于以上概念和性质，算法的基本步骤介绍于下。静态类型检查法的基本过程为：

a. 建立类属函数的混淆集；

b. 构造各个混淆集的顶集；

c. 对每个类属函数调用找出能覆盖它的所有混淆集；

d. 进一步检查 C 中得到的混淆集是否覆盖类属函数调用，从而确定一个混淆集；

e. 根据方法的具体性，保留 d 中所确定的混淆集的一个子集，这个子集就是动态分派时所要查找的集合。

1.3 **性能分析** 静态类型检查法是一种非常量时间算法，在最坏的情况下，其分派的时间复杂度是 $O(\|M\|\times n)$，其中 $\|M\|$ 为混淆集中元素个数，n 为类属函数的元数。但通常没有这么大的时间耗费。这种算法所获得的目标代码所占用的运行空间较小。因此，总的说来，静态类型检查法是对时间和空间性能的一种较好折衷解决办法。但是这种算法难于用于高要求的实时系统。

2. 压缩分配表法

E. Amiel 等在文[4]中提出了一种使用压缩的分派表优化多方法分派的算法。

2.1 **算法思想** 这个算法的目标是：①删除分派表中的空行和空列；②合并相同的行和列。图 5 所示的分派表经压缩后得到：

	{B}	{C}	{D,E}
{A}	1	1	1
{B,D}	—	2	2
{C,E}	—	2	3

图 7

为了能在运行时根据参数类型进行动态分派，我们还要为每个类型在各维上分配一个索引。比如上面的例子中第一维中，A 的索引为 1，B，D 的索引为 2，C，E 的索引为 3。这样，在分派时，首先根据参数查找索引表，再根据所得索引在分派表中查找 MSA 方法。

2.2 **算法步骤** 首先介绍一些基本定义。

· 第 i 维静态参数 类属函数 m 的第 i 维静态参数 $Static_m^i$ 的定义是：$Static_m^i=(T\mid\exists\ m_k,T_k^i=T)$。

· 类型覆盖 类型 T 的覆盖 cover(T) 的定义是：$cover(T)=(T'\mid T'\leqslant T)$。

· 第 i 维动态参数 类属函数 m 的第 i 维动态参数 $Dynamic_m^i$ 的定义是：$Dynamic_m^i=cover(Static_m^i)$。令 $Dynamic_m$ 表示为第 i($1\leqslant i\leqslant n$)维动态参数的叉乘，即 $Dynamic_m=\prod_{i=1}^{n}Dynamic_m^i$。

· I-Pole 称类型 T 是类属函数 m 的一个 i-pole(表示为 $is\text{-}pole_m^i(T)$)，当且仅当($T\in Static_m^i$)或者($\exists\ T_1,T_2\mid is\text{-}pole_m^i(T_1),is\text{-}pole_m^i(T_2),T_1<\!>T_2,T<T_1,T<T_2,\exists\ T_3\mid is\text{-}pole_m^i(T_3),T<T_3\wedge(T_3<T_1\vee T_3<T_2)$)。其中 $T_1<\!>T_2$ 表示 $T_1\not\leqslant T_2$ 且 $T_2\not\leqslant T_1$。令 $Pole_m^i$ 表示集合 $\{T\mid is\text{-}pole_m^i(T)=true\}$。

· I-Ambiguity 类属函数 m 的两个 i-pole T_1 和 T_2 是关于 T 的 i-ambiguous，当且仅当 $T_1<\!>T_2$，$T<T_1,T<T_2,\exists\ T_3\in Pole_m^i\mid T<T_3$ 且($T_3<T_1$ 或 $T_3<T_2$)。

· I-Influence 类属函数 m 的一个 i-pole T_p 的 i-influence(表示为 $influence_m^i(T_p)$)定义为：$influence_m^i(T_p)=\{T\leqslant T_p\mid\exists\ T'_p\in Pole_m^i,T\leqslant T'_p<T_p$ 或 $T_p,T'_p ambiguous_m^i T\}$。

关于 i-influence 有一个重要的性质：

定理 3 令 $Pole_m^i=\{T_p^1,\cdots,T_p^k\}$ 则 $\{influence_m^i(T_p^1),\cdots,influence(T_p^k)\}$ 是 $Dynamic_m^i$ 的一个划分。

进一步，可以证明两个重要的定理。

定理 4 对类属函数 m，类型 T 在第 i 维可被删除，如果满足条件：T 不属于任何一个的 i-pole 的 i-influence。

定理 5 类型 $T \in Dynamic_m^i$ 可在第 i 维合并到 $Pole_m^i(T)$ 中去，

下面给出算法的基本步骤：

a. 计算 i-pole 和 i-influence。

b. 为 i-influence 分配索引。

c. 为每个类型 T 存储其所属的 i-influence 的索引。

d. 为每个 $(T_1,\cdots,T_n) \in \prod_{i=1}^{n} Pole_m^i$ 确定 MSA(m($T_1,\cdots,T_n$))，填充分派表。

2.3 性能分析 压缩分派表法是一种常量时间算法，与未压缩的分派表相比，使用压缩分派表在分派时增加了 n(类属函数的元)次类型索引的查找。由于 n 一般在 2—4 之间，所以其分派的时间性能是良好的。

E. Amiel 等人认为，该压缩算法在多数情况下能得到最小的分派表，即完全实现算法的两大目标。然而，如何改进算法以保证在所有情况下都获得最小的分派表仍是一个亟待解决的问题。表格压缩的比例与类(及方法)的结构有紧密关系，因而还有两方面工作有待于进一步探索：压缩算法对现有的应用系统效果如何？从理论上说，压缩算法适合于什么样的类(及方法)的结构？

3. 自动机法

W. Chen 等人提出了一种用自动机实现动态分派的方法[22]。与静态类型检查法类似，该算法也是以划分混淆集为出发点的。

3.1 算法思想 此算法引入了查找自动机 LUA(lookup automaton)。它是一个五元组(Q,Σ,δ,q_0,F)，其中：Q 是有限状态集；Σ 是有限输入符号集(此处的输入符号就是类型)；δ 是状态转移函数(state transition function)；q_0 是起始状态；F 是终结状态集。在动态分派时，把参数作为输入符号输入 LUA，到达终结状态时所对应的方法即为 MSA 方法。

在实现自动机的过程中面临着这样一个问题：在 LUA 的一个状态下，是否为所有的合法输入都设定一个转移函数？回答是否定的。因为如果这样做，LUA 所占的空间是难以令人接受的。那么应该为哪些输入设定转移函数呢？对那些未设定转移函数的输入应如何处理呢？W. Chen 在文[22]中提出了一种解决办法。

3.2 算法步骤 首先介绍几个基本概念。

· 最大下界(greatest lower bounds) 类型 s 和 t 的最大下界 GLB(s,t)定义为(式中 T 为类型集合)：$GLB(s,t)=\{u \in T \mid u \leqslant s, u \leqslant t$，且$\nexists\ u' \in T$ 使得 $u \prec u' \leqslant s$ 且 $u \prec u' \leqslant t\}$

· 封闭性 对于 $S \subseteq T$，如果$\forall\ s,t \in S$，$GLB(S,t) \subseteq S$，我们称 S 是封闭的(colsed)。闭包(closure)$S \subseteq T$ 的闭包 closure(S)定义为所有封闭的集合 T($S \subseteq T \subseteq T$)的交集。显然闭包是封闭的。

· 最小上界(least upper bounds) 对 $t \in T$，$S \subseteq T$，t 关于 S 的最小上界 LUB(t,s)定义为：$LUB(t,s)=\{S \in s \mid t \leqslant s$，$\nexists\ s' \in S$ 使得 $t \leqslant s' \prec s\}$

定理 6 $S \subseteq T$ 封闭的充要条件是$\forall\ t \in T$，LUB(t,S)最多包含一个元素。

一个 n 元类属函数的 LUA 如下构造：

for i=0 to n−1 do

a. 构造第 i 级状态(第 0 级状态即为初态)，并保存可应用的方法及方法的优先顺序(初态的可应用方法就是该类属函数所有的方法)

b. 对所有的第 i 级状态分别执行，令 S 为当前状态所有可应用方法的第 i+1 维参数类型的集合，为 closure(S)中每一个元素设定转移函数(通过这些转移函数就可获得第 i+1 级状态)。

在动态分派时，如果没有对应于某个输入的转移函数，我们就可以寻找其最接近的超类(根据定理 6，这样的超类最多只有一个)，如果找到，则使用对应的转移函数，否则就可判定为非法函数调用。

3.3 性能分析 自动机算法实现了常量时间动态分派。一般说来，该算法所生成的 LUA 所占用的运行空间比分派表要小。不足之处是该算法本身的实现较为复杂，另外，由于算法的核心 closure 的计算依赖于整个类型结构及混淆集的信息，破坏了模块编译的独立性。

4. 小结

上面谈到的三种算法是目前研究工作中较有成效的代表。当然，它们都存在着不同程度的缺陷与不足，有待于进一步完善。

总的说来，静态类型检查算法所生成的目标代码占运行空间最小，自动机法次之。而从目标代码的时间性能来看，分派表最快，自动机法次之。要对三种算法作出合理的评价，还需要做大量的实验工作。

四、其他工作及未来展望

在多方法分派的研究中，还有许多其他的工作。

(下转第 9 页)

参考文献

[1] P. Henderson, Purely functional operating systems, In J. Darlington et al. eds., Functional Programming and its Applications, Cambridge University Press, 1982

[2] JT O'Donnell, Dialogue: a basis for constructing programming envirnoments, In Proc ACM Symposium on Language Issues in Programming Environments, Seattle, ACM, 1985

[3] E Moggi, Notions of computation and monads, Information and Computation, 93, 1991

[4] SL Peyton Jones & PL Wadler, Imperative functional programming, In Proc. 20th ACM Symp. on Principles of Programming Languages. Charleston, 1993

[5] PM Achten & MJ Plasmeijer, The ins and outs of Clean I/O, J. of Functional Programming 5(1), 1995

[6] 袁华强，函数式 I/O 系统的研究与实现：一种 Monad 方法[博士学位论文]. 上海交通大学, 1996

（上接第 5 页）

D. Moon[20]提出了用散列表技术的查找结构，并将其应用于 Flavor 系统中。在散列表中要预先填充所有可能的分派情况，而在多方法环境下，各种分派情况的总数是组合爆炸的（复杂度为 $O(e^n)$[21]）。

G. Kicales[6]也提出了类似的方法应用于 CLOS 系统中。作为改进，P. H. Dussud[21]提出使用动态缓冲技术，动态缓冲中只保存被调用的方法。这种技术是非常量时间的，也没有解决组合爆炸的问题。

C. Lecluse 等[22]提出了结构子类化（sturctural subtyping）的概念。他们要求定义辅助的方法以满足仅仅通过子类型关系就可确定 MSA 方法。

W. G. Mugridge 等[23]定义参数序列 $(t_1, \cdots, t_n)$ 的覆盖（cover）为集合 $\{(S_1, \cdots, S_n) \mid S_1 \leqslant t_1, \cdots, S_n \leqslant t_n\}$，一个方法的可应用性定义为方法参数的覆盖与实际调用参数的覆盖的交集（可以认为集合越小的方法越具体）。对于一个类属函数调用，可通过比较相应覆盖的交集来确定 MSA 方法。这种解决方式应用于 Kea 中，当两个方法无法确定优先顺序时就声明调用是二义的。

总的说来，目前的研究工作都着重于提高各自算法所得到的目标代码的时间和空间性能。这些算法各有优劣之处，其中一部分已经应用于实际的多方法 OO 语言中。

我们认为，未来在多方法分派研究中的工作主要有几个方面：①对目前算法的改进和完善；②提出新的综合性能更佳的算法；③通过理论分析和实践检验，综合评价各种算法；④对算法进行综合。

对算法进行综合也有不同的方法。一方面，我们可以在编译系统中设置编译开关，由程序人员根据实际情况在时间与空间性能之间进行权衡和选择。另一方面，也是更重要的一方面，我们可以让编译系统检查分析程序的结构，自动选择最佳算法，甚至组合各种算法。

由于多方法语言强大功能所具有的吸引力，这方面的研究工作将会进一步深入下去。尽管存在着相当大的困难，我们仍然可以期望，在不久的将来，多方法语言将获得普遍的接受和应用。（参考文献共 22 篇略）