函数式元编程语言的设计要素*)

欧阳坚　罗晓光　王生原　戴桂兰　张素琴
(清华大学计算机系　北京 100084)

摘　要　本文介绍基于函数式语言的元编程系统,讨论元编程系统特别是同构系统的语言特点。从程序反射的角度分析元编程系统对程序设计语言在自我表示、自我分析和控制等方面的要求。以 MetaML 和 Template Haskell 为例论述在函数式语言中为了支持元编程需要扩展的机制,包括语法、语义、类型系统、安全的变量使用等,以及它们的实现方案、各方案的特点。最后总结一些元编程系统的共同点,并预测未来的发展趋势。

关键词　函数式语言,程序反射,元编程系统,同构系统

The Design Elements of Functional Meta-programming Languages

OU-Yang Jian　LUO Xiao-Guang　WANG Sheng-Yuan　DAI Gui-Lan　ZHANG Su-Qin
(Department of Computer Science and Technology, Tsinghua University, Beijing 100084)

Abstract　This paper firstly gives a brief introduction to functional meta-programming systems and presents the language sugar of meta-programming systems especially homogeneous systems. Meta-programming languages' extra requirements including self expressing, self reasoning, and self controlling in meta-programming systems are analyzed in a view of procedural reflection. Then the paper discusses the issues of expanding a normal language to a meta-language, such as syntax, semantics, type system, hygienic variable usage and their implementation choices, by using MetaML and Template Haskell as examples. Finally, it sums up with common characteristics of meta-programming systems and predicts future trends.

Keywords　Functional language, Procedural reflection, Meta-programming system, Homogeneous systems

随着软件技术的发展和程序设计语言的演进,以程序作为处理对象的情况越来越多,编写这样的处理程序叫元编程(meta-programming)。元编程常用来克服传统程序设计的局限性,例如语言的表达能力、程序的灵活性及程序执行的效率等[1]。

元程序(metaprogram)是控制别的程序的程序。在元编程系统(meta-programming system)[2]中,元程序控制目标程序(object-program),包括构造目标程序、组合目标程序观察目标程序的结构和其他属性,执行目标程序并获取结果等[3]。编译器,部分计算程序(partial evaluators)、分析程序生成器(parser generators)都是元程序。

函数式程序设计[4]是面向问题的程序设计方法。函数式程序设计语言具有较好的数学性质和高度的抽象性,并且正确性容易验证,因此很适合作为元编程系统的编程语言。现在很多函数式语言都在一定程度上有元编程的支持或发展出专用于元编程的方言。它们在自动程序翻译、程序自动分析和生成、定理证明与推理及 MSP[1,5]、DSL[6]、RAP[7]等方面都重要的应用价值。

本文主要讨论使用函数式语言的同构元编程系统。以两种较有影响力的实验性函数式元编程系统 MetaML[8] 和 Template Haskell[9]为例,从程序反射的角度分析在函数式语言中为了实现对元编程的支持需要扩展的语言成分及其实现等问题。

第 1 部分介绍同构元编程系统的优点,第 2 部分浅析元编程与反射的关系,第 3 部分先总结元编程语言的要点和基本概念,然后以 MetaML 和 Template Haskell 为例进一步分析元编程需要的机制及可能的实现方法,最后给出结论并展望发展趋势。

1　同构元编程系统

元编程系统分为两种:同构系统(homogeneous systems)和异构(heterogeneous systems)系统。同构系统中元语言和目标语言是同一种语言,比如 MetaML;异构系统中元语言和目标语言是不同的语言,比如 YACC。两种系统对于自动程序分析都是很有用的,但是同构系统具有一些异构系统没有的优势[3,10]:

- 只有同构系统可以是多层的,目标程序本身也可以是元程序,即它是控制另一层目标程序的程序。
- 只有在同构系统中才能只用一个类型系统(type system)约束元程序和目标程序。
- 只有同构系统才能支持反射,提供程序到其表示的转换操作,运行时能生成代码。
- 在同构元编程系统中由于目标语言和元语言使用相同的表示,因此它们能共用一个分析程序(parser)、类型检查程序(type checker)等,实现资源的复用。
- 同构系统中用户只需要学习一种语言。

2　元编程与反射的关系

在同构的元编程系统中,元编程和反射能力是两个既有联系又有区别的概念。元编程强调对程序的分析和构造,目

*)国家自然科学基金项目(编号:#60403022)。欧阳坚　硕士研究生;罗晓光　硕士研究生。

的是提高程序执行的效率，增强程序的表达能力等。程序反射指用推理过程显式地观察、表示和控制对程序的理解，并且这个推理过程本身也能用这种能力表示；它强调程序的自我观察，思考自身的状态，并能自我调节的能力。反射是元编程语言的一个重要特点。

3-Lisp 的作者 Smith 在文[10,11]中用一个解释器的无限塔模型来描述反射系统，其中每一个解释器是能被它的上一层解释器所理解的程序，用户程序运行在最底层。为了实现反射机制这个塔结构允许跨相邻层的操作(level shifting operators)，即在第 n 层执行的程序能够通过跨层操作用第 $n+1$ 层的解释器来观察它的执行，这样的跨层操作称为具体化(reification)，具体化的逆操作叫做反射(reflection)[10~12]。

这种塔模型与同构元编程系统不谋而合：在同构元编程系统中元程序和目标程序是同一种语言，这种语言能表示自己、分析自己，并且 n 层的目标程序可以是控制 $n+1$ 层的元程序，因此具有自然的层次结构。这种同构元编程语言可以看成一种反射式语言，适合用来构造十分灵活的系统。从反射的角度来看，同构元编程语言与普通语言最大的不同也就是它们加入了一些反射的语义，并增强了对目标代码的处理能力。

3 同构元编程语言

可以认为，同构元编程系统中的元编程语言是在不具备元编程机制的程序设计语言的基础上扩展反射机制得到的。加入的主要特征有：

• 表示方法。为了提供控制、分析、构造元语言的能力，首先要能在系统中表示目标程序，在同构系统中就是用一种语言表示其自身。常用的表示方法有字符串、记录、对象等。在函数式语言的系统中常用代数数据结构(algebraic data-structures)[3]来表示，它比起使用简单的字符串更易于分析和控制。

• 表示函数。确定目标程序的表示方法后接下来的问题就是如何构造程序的表示，即如何得到一段目标程序的表示。元编程系统的目的是为了方便对目标程序的操作，因此提供了一种简洁的语法机制：表示函数(representation function)[13]——由程序到程序的表式结构的映射。表示函数的反函数也是需要的最基本操作，因为通常对目标程序的分析和构造是为了得到新的目标程序，需要把结构重新映射成程序。实际的元编程语言为了方便程序员，除了加入这两个操作，还会加入一些其他操作，方便用户使用。

除了提供实现元编程上述基本机制，系统还需要提供发现错误的能力，以便于调试和维护。错误包括元层程序的错误和目标层程序的错误。

• 语法正确。用代数数据结构而不是字符串来表示目标程序的另一个优点是只要保证元程序的类型正确，就能保证目标程序的语法正确。

• 类型系统。在静态类型语言中，我们不仅希望目标程序的语法要正确，也希望它的类型也是正确的，即只要程序通过了编译器的检查，元程序和目标程序在运行时都不会出现类型错误。这个要求在 MetaML 和 Template Haskell 都得到了实现。

3.1 MetaML 与 Template Haskell

ML 和 Haskell 是当前最有影响力的两种函数式语言。为了提供对元编程的支持，它们分别发展出了 MetaML 和 Template Haskell 两种方言。

MetaML 由 CRML[14,15]发展而来，是为了多阶段程序设计(multi-stage programming)[1,5,16,17]而设计，而且在其基础上结合面向对象的 ML 方言 OCaml[18]又发展出了方言 MetaOCaml[19,20]。为了简洁和统一本文中以 MetaML 为例来说明这系列语言的特点，MetaML 类似 SML，不过增加了多阶段编程的构造符，在同一个编程范例中紧密地集成了程序的构造、结合、编译和执行操作[1,17]。

Template Haskell 是对 Haskell 语言的扩展，为其增加了编译时元编程的机制，目的是为了让程序员能够在不改变编译器的条件下定义新的语言特征。

虽然在很多方面 MetaML 和 Template Haskell 的侧重点有所不同，但是 MetaML 和 Template Haskell 都比较好地提供了元编程系统的功能。后面的论述中主要使用 MetaML 和 Template Haskell 来举例说明元编程语言的特点。

3.2 表示目标代码

使用字符串来表示目标程序，虽然容易构造和组合，但是分解和分析却十分困难，因此元编程系统普遍采用具有一定结构的数据形式来表示目标程序。函数式语言的元编程系统中，目标程序通常用代数数据结构来表示，并且有用户不可见和用户可见之分。

不可见代数数据结构对用户隐藏程序表示的具体细节，用户通过系统提供的一组接口间接完成对元程序的操作，不需要知道数据结构的具体实现，MetaML 是此类型的系统。可见，代数数据结构对用户开放程序表示的具体细节，用户既能通过系统提供的一组接口间接完成对元程序的操作，也可以直接对数据结构进行操作，Template Haskell 是这种类型的系统。

在不隐藏目标代码具体实现的系统中，用户能自己定义函数或者直接控制，因此在 Template Haskell 中用户也能自己写程序，完成跨层操作。实际上，Template Haskell 为目标程序提供了三个层次的表示方法[21]，这三层方法的易用性递增但是表达能力递减：

(1)底层包括两部分：一是 Haskell 程序的代数数据类型(algebraic data types)的表示；二是 quotation monad(Q)，它封装了产生新的名字、失败处理和 I/O 操作。

(2)语法构造函数库，例如 Tup 和 App 等，为访问底层的结构提供了方便的方法。

(3)quasi-quote 的方式，这是构造目标程序的最简单方法，但是有一些重要的元程序它是无法表达的。

Template Haskell 中，程序员可以自由地混合使用这三层表示，使用后两层比第一层简便。

3.3 元编程的基本操作

函数式语言通常提供一种简单的 quotation/quasi-quote 语法来实现用户与目标程序的接口--表示函数，即通过一个运算符把程序括映射为表示程序的数据结构，并且提供 anti-quotation 的方法来实现 quotation 的逆操作。

MetaML 中，对应 quotation 的操作是 Meta-Brackets("〈_〉")。尖括号内是一段代码，整个表达式表示括号内的代码的表示结构。如〈2+3〉表示 2+3 这个表达式的数据结构，即在这里 2+3 是目标层程序，〈2+3〉是它对应的元层结构，因此它与(5)是不同的(〈5〉表示 5 的表示结构)。在多阶段程序设计中，尖括号表达式不仅表示一段目标程序，也是推迟计算的主要手段[1,16,17]。

Template Haskell 中，对应 quotation 的操作是 quasi-quote brackets("[| _ |]")。与 MetaML 类似，"[|"和"|]"之间是一个表达式，结果是表示这个表达式的数据结构。不同的是，MetaML 隐藏了这个数据结构的实现。Template Haskell 中，这个结构也是普通的 Haskell 数据类型，名为 Expr，用户可以直接对其进行各种操作[21]。

anti-quotation 通常用来把小的目标程序组合成较大的目标程序，完成 quotation 的逆操作。

MetaML 中对应的 anti-quotation 操作是 Escape。Escape("~")，用于把小程序段组合成大块的程序段，即它是 Meta-Brackets 的逆操作。但不同的是，它只能出现在〈〉中。比如〈~〈2+3〉+~〈5〉〉的结果是〈2+3+5〉。

Template Haskell 中，对应的 anti-quotation 操作是 Splice annotation("$")。由于 Template Haskell 是一个编译时元编程(compile-time meta-programming)系统，因此 Template Haskell 所有的元程序都是在编译的时候执行，在执行时所运行的程序是由元程序在编译时执行所产生的普通 Haskell 程序。"$"告诉编译器在编译时计算后面的元程序表达式(类型为 Expr)，并返回其结果。

模式匹配(pattern matching)是函数式语言分析数据结构的非常方便的手段，很多元编程系统对模式匹配也进行了扩展，使其支持对目标代码数据足够的匹配，即能够匹配 quotation 和 anti-quotation 操作符。

例如 MetaML 里的下面一段程序：

```
-| fun f 〈fn x⇒~(g 〈x〉)+3 〉
= 〈fn y⇒~(g 〈y〉)〉
| f x = x;
val f = Fn : ['a]. 〈'a → int〉→ 〈'a → int〉
-| f 〈fn x ⇒ x + 1〉;
val it =
〈(fn a ⇒ a %+ 1)〉
: 〈int → int〉
```

3.4 层次(阶段)

由于一段目标程序可能是控制另一段目标程序的元程序，自然地产生了层次(也叫阶段)的概念，即在不同的层次对同一段程序有不同的理解。通常，最外层的程序被规定为第 0 层，而一个 quasi-quote 操作则会把其内部代码的层次增加 1。而 anti-quotation 则会把其作用的代码的层次减少 1。简单地说，一段代码的层次就是作用于它的 quasi-quote 操作次数减去作用于它的 anti-quotation 次数。例如在 MetaML 程序 fun $f n = \langle fn\ x \Rightarrow \langle \sim x + n \rangle \rangle$ 中：整个表达式处于 0 层，函数 fn $x \Rightarrow \langle \sim x + n \rangle$ 处于 1 层，$\sim x + n$ 处于第 2 层，x 则处于第 1 层。容易看出，$n+1$ 层的程序是第 n 层的目标程序同时又是第 $n+2$ 层的元程序。在 Template Haskell 中也有类似的概念，比如 $g\ x = \$(h\ [|x * 2|])$ 中：h 处于 -1 层，$x * 2$ 处于 0 层。MetaML 与 Template Haskell 不同的是：MetaML 中 0 层的代码不能再进行 Escape 操作，run(见 3.5)则可以。例如：

```
-| fun f x = 〈x〉;
val f = Fn :['a]. 'a→〈'a 〉
-|~(f 3);
Error: level-0 code cannot be escaped
-| run(f 3);
val it = 3 : int
```

而 Template Haskell 则允许在类型正确的情况下任何层都能进行 $ 操作。

```
f Int → ExpQ
f 1 =::[| "11" |]
f 2 =[| "22" |]
f x =[| "unknown" |]
$(f 7)则可以得到"unknown"
```

跨阶段保持性(cross-stage persistence)：在灵活的元编程系统中高层次的程序应该可以使用比其低层的程序中的绑定和函数，这样使较低层的程序得到较好的复用。即第 $n+1$ 层的程序应该能使用在 1 到 n 层程序中定义的绑定和函数。好的元编程语言应该有这样的语义，提高程序的复用性。

跨阶段安全性(cross-stage safety)：在安全的元编程系统中，低层次的程序应该不能使用比其高层的程序中的绑定和函数，即第 $n+1$ 层的程序是第 n 层程序控制的目标程序。因此在 $n+1$ 层程序中的绑定和函数不应该在层次比 $n+1$ 层低的程序中被使用，因为在这些层的程序中它们是没有定义的。安全的元编程系统应该能发现这样的错误。例如在 MetaML 程序 $fn\ a \Rightarrow \langle fn\ b \Rightarrow \sim(a+b) \rangle$ 中，b 是在第 1 层中定义的，它应该只能被大于等于 1 层的程序所使用，但是 $a+b$ 中的 b 处于第 0 层，因此这是一个看上去可行、实际上并不正确的程序。

3.5 从反射的角度看跨层操作

反射是同构元编程系统的一个重要特点。反射塔模型中允许位于某层上的程序通过跨层操作访问表示自己的数据结构，即一个在 n 层运行的程序能够通过跨层操作从第 $n+1$ 层的解释器的角度观察自己的执行。这样的操作叫做 reification，其逆操作叫做 reflection。函数式同构元编程系统中的 quotation 和 anti-quotation 就是最常见的跨层操作，它们分别完成 reification 和 reflection 的功能。

MetaML 和 Template Haskell 中常见的跨层操作有

· 类 reification 操作

MetaML 中类似 reification 的操作有两个 meta-brackets "〈 _ 〉"和 lift，lift 与 meta-brackets 功能类似，也是完成从 n 层到 $n+1$ 层的变化。不过与 meta-brackets 直接把程序提升到 $n+1$ 层不同，lift 先在 n 层执行程序，然后把得到的结果提升到 $n+1$ 层。比如 lift(2+3)的结果是〈5〉，因此 lift 的功能类似于函数 $fn\ x \Rightarrow \langle x \rangle$，但是 lift 不能作用于函数 Template Haskell 中类似 reification 的操作有 quasi-quote brackets"[| _ |]"和 Reification，Reification 允许程序员获得一些程序结构即属性(声明、类型、运算优先级、语句所处的位置等)的表示，便于加以分析和控制。

· 类 reflection 操作

MetaML 中类似 reflection 的操作也有两个 Escape"~"和 run。run 完成从 $n+1$ 层到 n 层的变化，不过先运行目标程序，然后把执行的结果降到第 n 层。例如 $run\ \langle 2+3 \rangle$ 的结果是 5。Template Haskell 中类似 reflection 的操作有 splice annotation"$"和 splice，"$"告诉编译器在编译时计算后面的元程序表达式(类型为 Expr)，并返回其结果。splice 功能与"$"类似，不过它是用于对 Reification 产生的结果进行操作。

3.6 类型系统

静态类型(强类型)程序设计语言中，类型系统的作用在于提前发现程序的错误和改善程序执行时的性能。对于元编程系统中目标程序也存在同样的问题。元编程系统不仅希望

保证目标程序的语法没有错误，也希望保证目标程序的类型正确。同构元编程系统的一个优点就是能只用一个类型系统约束元程序和目标程序，因此类型系统也是同构元编程系统的一大特色。

尽管 Haskell 和 ML 都不用显式地声明变量类型，它们仍是静态类型的语言。编译时通过类型推导(type inference)，确定所有没有显式声明的变量类型(函数可以是多态的)，并进行类型检查。Template Haskell 和 MetaML 同样也继承了这一特点。而 Scheme[22](Lisp 的一种方言)是动态类型(弱类型)的语言中，类型与值(对象)相关联而不与变量相关联，因此 Scheme 能通过宏实现语言的自我表示和控制，不需要进行静态类型检查，更不需要为元编程建立新的类型系统。

静态类型的类型检查主要是编译时确定各层程序数据结构的类型，并发现与类型不一致的操作。类型系统首先要确定表示目标程序的数据结构的类型，通常有两种不同的方式：

一种是所有的目标程序拥有同一个类型，比如 Template Haskell 和一种在 Intel 用于硬件设计的反射式函数式语言 reFLect[23]。在 Template Haskell 中，所有形如"[| |]"都具有相同的类型 Expr[21, 24]，Expr 是实现目标程序表示的计算类型－单体(monad)[25]。如[| 4+5 |]和[| True |]等都拥有同一个类型 Expr。reFLect 也类似，〈〈x+y〉〉与〈〈true〉〉也都有同样的类型 term。两者不同的是 Template Haskell 在编译时就完成所有元程序的控制，所有形如"[| |]"的表达式都会在编译时打开，最后得到普通的 Haskell 程序，因此 Template Haskell 能在编译时保证所有层次的目标程序的类型正确。而 reFLect 在编译时只完成最外层的类型检查，因此在执行时还需要进行类型检查，并且可能在执行时发现目标程序的类型错误。例如，虽然[| 1+5 |]与[| True || False |]的类型都是 Expr，但是表达式 $([| 1+5 |])+$([| True || False |])在编译时就会被替换成 6+True，然后发现类型错误。但是在 reFLect 中〈〈1+5〉〉和〈〈true〉〉类型都是 term，表达式〈〈~〈〈1+5〉〉+~〈〈true〉〉〉〉在编译时只在最外层做类型检查，则认为它是类型正确的表达式。~〈〈1+5〉〉和~〈〈True〉〉在执行时会被计算成 6 和 True，并且在执行时类型检查发现 6+True 的类型错误。

另一种方式是让不同层次和类型的目标程序的数据结构具有不同的类型，比如 MetaML[1, 19, 24]。在 MetaML 中，形如"〈 〉"的表达式的类型与"〈"和"〉"之间的表达式类型有关，比如〈1+2〉有类型〈int〉，〈not true〉有类型〈bool〉，〈〈 5 〉〉有类型〈〈int〉〉，〈fn n ⇒ 〈n + 1〉〉有类型〈int → 〈int〉〉。因此，MetaML 能在编译时检查所有层的程序的类型，比如〈~〈4+2〉 + 9 〉虽然~〈4+2〉在执行时会被计算成 4+2，但是由于〈4+2〉具有类型〈int〉，因此在编译时就能知道~〈4+2〉具有 int 类型，就能确定~〈4+2〉 + 9 是类型正确的表达式。

3.7 静态作用域与名字冲突

元编程语言的变量通常采用静态作用域，它能便于程序的理解和减少错误的发生。静态作用域指：每个变量的引用都被绑定到模板展开前变量原始定义初始环境中，而不是在变量使用的环境中。它能避免同一个变量在不同的地方使用得到不同的效果，减少不小心的错误。静态作用域还有一个作用，就是防止变量名字的意外冲突。

名字意外冲突(accidental name capture)来源于 λ 表达式的约束变量与自由变量同名的情况，约束变量应该可以通过 α 变换替换成其他不与自由变量相同的名字。元编程语言需要在产生时提供一种约束变量换名机制，以防止错误的产生，即提供一种卫生的(hygienic)变量使用机制。这种机制对于类似 Scheme 这样的弱类型语言实现基于宏的程序反射结果也同样十分重要。

例如考虑 MetaML 函数 fun t e = 〈fn x ⇒ x +~e〉，我们希望 t 〈1+5〉的结果是〈fn x ⇒ x +(1+5)〉，同时 t 〈1+x〉却不能是〈fn x ⇒ x+(1+x)〉，因为这里绑定的 x 只是 x +(1+x)中的第一个 x，它是可以被任意改变名字的。因此我们希望 t 〈1+x〉的结果是〈fn y ⇒y+(1+x)〉。例如下面的 MetaML 程序：

```
-| val a = 8;
val a = 8 : int
-| fun t e = 〈fn x ⇒ x +~e〉;
val t = Fn :['a#].〈'a〉→〈'a →'a〉
-| t 〈1+a〉;
val it =
〈(fn a ⇒ a %+ 1 %+ %a)〉
:〈int→int〉
-|(run it)3;
val it = 12 : int
```

可以看出〈(fn a ⇒ a%+ 1%+ %a)〉中的%a 与约束的 a 是不同的。

由于 MetaML 中变量是完全静态作用域的，意外的名字冲突可以得到避免[1, 16]。

又例如 Template Haskell 程序[21, 24]：

```
cross2a:: Expr →Expr → Expr
cross2a f g = [| \(x,y)→($f x, $g y)|]
```

我们希望在执行 cross2a(var "x")(var "y")结果不会是\(x,y)→(x x,y y)，而是约束变量(x,y)被自动地替换为别的名字，实际上 Template Haskell 的 quasi-quotes 提供了这种功能。运行 cross2a(var "x")(var "y")的结果是\(x0,y1)→(x x0,y y1)。可以看出，同样 Template Haskell 也采用了静态作用域的语义。

在不隐藏目标代码数据结构的系统中，应该为用户提供产生唯一的名字的机制(通常是一个叫 gensym[3, 21]的函数)，它需要维护一些状态即副作用(side effect)。Expr 实现为 Haskell 中的 Monad，很好地实现了 gensym 需要的副作用。例如[21]：

```
cross2c :: Expr →Expr → Expr
cross2c f g =
do { x←gensym "x"
; y ← gensym "y"
; ft ←f
; gt←g
; return(Lam [Ptup [Pvar x,Pvar y]]
(Tup[App ft(Var x)
,App gt(Var y)]))
}
```

4 元程序执行的时机

从前面 MetaML 和 Template Haskell 的不同可以看出，元编程系统有两种差异很大的实现方式：运行时元编程(runtime metaprogramming)和编译时元编程(compile-time

metaprogramming)。这两种不同最大的区别是对目标程序的构造、分析、控制等的时机不同。前者是在程序实际运行时完成的,后者是在程序编辑阶段完成的。采用不同的方式是因为设计系统的目的不同。

MetaML是一个运行时元编程系统,为产生多阶段程序设计而设计,主要目的是提高程序针对具体问题的性能。Template Haskell是一个编译时元编程系统,为Haskell语言增加了编译时元编程机制,目的是为了让程序员能够在不改变编译器的条件下为定义新的语言特征,增强了语言的表达能力。

Template Haskell与C++模板元编程类似,都是在编译的时候执行。在执行时所运行的程序是由元程序在编译时执行所产生的普通程序,然后再编译一次,得到可执行程序。但是它们也有所不同,C++ Templates的发明是为了支持generic programming,其静态计算(static computations)和代码产生的功能是意外的发现,因此用C++ Templates进行元编程并不是很自然,需要一些技巧[26]。与之不同,Template Haskell是专门为元编程而设计的,元元语言采用Haskell本身,并增加了一些语法构造符和库函数,表达能力更强,更简洁易用。

总结和发展趋势 元编程系统的目的是为程序员提供方便、安全的构造和控制程序的手段。它把一些构造和控制程序的普遍的操作抽象出来实现,并添加到系统中,提供给程序员使用。这样,既可以减少程序员的重复劳动,又容易保证程序的正确性。因此提供目标程序简单的、自然的、表达能力丰富的表示方法,是元编程系统的主要问题。从MetaML和Template Haskell的共同点可以看出一些对元编程系统的需要:好的元编程系统应该提供友好的人和系统交互的接口。除提供目标程序的构造、目标程序的组合的基本能力以外,还应该有合理的类型系统,保证安全的变量使用,并且能够执行目标程序,能够输出目标程序,观察和分析目标程序[3]。

程序设计语言一直处于不断的发展中,对程序的控制从宏到模板再到程序自身,越来越方便,越来越安全。随着新的应用和挑战不断地出现,今后的元编程的应用将越来越丰富,元编程系统也会向有更灵活的语义、更强的表达能力、更模块化、更丰富的库函数方向发展。

参考文献

1 Walid T. Multi-Stage Programming: Its Theory and Applications. Oregon Graduate Institute School of Science \ Engineering, 1999
2 Robert D C, Ito M R. Grammar-Based Definition of Metaprogramming Systems. ACM Trans Program Lang Syst, 1984, 6: 20~54
3 Tim S. Accomplishments and Research Challenges in Meta-programming. Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation, Springer-Verlag, 2001
4 Hughes J. Why functional programming matters. Comput J, 1989, 32: 98~107
5 Multi-stage Programming Homepage: http://www.cs.rice.edu/~taha/MSP/
6 Czarnecki K, O'Donnell J, Striegnitz T, et al. DSL Implementation in MetaOCaml, Template Haskell, and C++. DSPG'04, 2004
7 Resource Aware Programming (RAP) Homepage: http://www.cs.rice.edu/~taha/RAP/
8 MetaML Home Page: http://www.cse.ogi.edu/PacSoft/projects/metaml
9 Template Haskell Homepage: http://www.haskell.org/th/
10 Cantwell S B. Reflection and Semantics in a Procedural Language: [PhD thesis]. MIT Laboratory for Computer Science, 1982
11 Cantwell S B. Reflection and semantics in LISP. Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. Salt Lake City, Utah, United States: ACM Press, 1984
12 Ming-Yuan Z. Computational reflection in PowerEpsilon. SIGPLAN Not, 1994, 29: 13~19
13 Anurag M, Daniel P F. Towards a Theory of Reflective Programming Languages. Informal Proceedings of the Third Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'93, 1993
14 Hook T S J. A Semantics of Compile-time Reflection. Dept of Computer Science and Engineering, Oregon Graduate Institute, Portland, Oregon Technical Report 93-019, 1993
15 Tim S. Guide to using CRML: Compile-Time Reflective ML
16 Tim S, Martel M. Introduction to Multi-Stage Programming Using MetaML Revision 2
17 Walid T, Tim S. Multi-stage programming with explicit annotations Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. Amsterdam, The Netherlands: ACM Press, 1997
18 The OCaml Language Homepage: http://www.ocaml.org/
19 MetaOCaml Homepage: http://www.metaocaml.org/
20 Cristiano C, Walid T, Liwen H, et al. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. Generative Programming and Component Engineering (GPCE'13), 2003
21 Tim S, Peyton J S. Template meta-programming for Haskell. SIGPLAN Not, 2002, 37: 60~75
22 Richard K, William C, Jonathan R. Revised5 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notices, 1998, 33: 26~76
23 Jim G, Tom M, John O I. A Reflective Functional Language for Hardware Design and Theorem Proving. European Joint Conferences on Theory and Practice of Software (ETAPS), 2004
24 Tim S, Peyton J S. Notes on Template Haskell Version 2. November 7 2003
25 Philip W. Monads for functional programming. Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School, 1993
26 Template Metaprograms: http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html

(上接第233页)

2 J2EE Connector Architecture Specification, version 1.5. http://www.jcp.org
3 Yee A, Apte A. Integrating your e-Business enterprise. Sams Publish, 2001
4 Mehta N R, Medvidovic N, Phadke S. Towards a taxonomy of software connectors. In: Proc. of the 22nd Intl. Conf. on Software Engineering, Limerick, Ireland, 2000. 178~187
5 Deline R. A catalog of techniques for resolving packaging mismatch. In: Mili A, Mittermeir R, eds. Proceedings of the Symposium on Software Reusability. New York: ACM Press, 1999
6 Papazoglou M P, Georgakopoulos D. Service-oriented computing. Communication of ACM, 2003, 46(10): 25~28
7 Gamma E, Helm R, Johnson R, et al. Design Patterns-Elements of Reusable Object-oriented software. Addison-Wesley Publish, 1995
8 Shaw M, Garlan D. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996
9 Barrett D J, Clarke L A, Tarr P L, et al. A framework for event-based software integration. ACM Transactions on Software Engineering and Methodology, 1996, 5(4): 378~421