arXiv:2303.10453v1 [cs.PL] 18 Mar 2023

逻辑编程中的模块化与分离编译

Steven Holte 和 Gopalan Nadathur

明尼苏达大学计算机科学与工程系，EE/CS 大楼 4-192 室，Union Street SE 200 号，明尼阿波利斯，MN 55455 电子邮件： {holte,gopalan}@cs.umn.edu，传真：612-625-0572

摘要。以模块化方式组合代码的能力对于构建大型程序至关重要。在逻辑编程环境中，理想的做法是通过基于逻辑的机制实现此类功能。本文介绍一种实现这一目标的方法，它简化并合理化了此前在 *Teyjus* λProlog 语言实现中支持的特性。在此方案中，模块对应于一个代码块，其外部视图由签名来调节。因此，签名施加了一种隐藏形式；在逻辑层面上，这可通过对谓词名、函数名和常量名进行存在量化来解释。模块交互基于 *accumulation*，这是一种将其中的子句定义合取起来的机制，同时会遵循签名解释所产生的存在量词作用域。我们的首要贡献是证明，这种用于静态构造命名空间的简单机制足以实现与代码作用域有关的特性，而此前人们认为需要通过动态控制谓词定义才能实现这些特性。我们提出的模块功能此前已在 *Teyjus* 系统中通过对累积的模块进行编译时内联来实现。这种方法的缺点是不支持分离编译。我们的第二项贡献是一种方案，可让每个不同的模块分别编译，并将内联任务推迟到后续的配套链接过程中。

1 引言

本文关注逻辑编程中的模块化处理。对这一特性的支持，对于该范式在重要应用中的实现至关重要。通过组合小型且定义明确的代码单元来开发系统，不仅是管理复杂性的关键，也有助于复用编程工作。此外，模块化开发为程序划定了边界；这些边界对于静态分析技术的实际应用可能十分重要，也是分离编译与测试这一概念的基础。鉴于此，逻辑编程中的模块化问题受到理论探讨 [6,7,10,14]，被纳入实际系统 [2,15,16] ，并成为围绕 Prolog 标准化讨论的主题 [5]，这并不令人意外。

模块化的概念通常通过跳出逻辑基础，引入用于组合独立构建的程序片段的元语言机制，融入逻辑编程系统。这种做法略显遗憾：逻辑编程的一大优势在于它以逻辑为基础，而只要选择得当，这一基础也可以用来推理代码单元之间的交互，

然而，也存在过于关注逻辑层面的风险。例如，早期基于逻辑的方法曾利用蕴含式或目标中的上下文来控制代码可用性 [7,10]。这些机制要求动态管理谓词定义，因此实现成本较高。此外，处理上下文的特定方式有时会导致程序行为有悖于对模块化的实际理解。

本文提出一种兼顾逻辑与实践考量的模块化处理方法。该方法借鉴了基于逻辑的方法 [8] ，这种方法应用于 Teyjus 实现 [12] 的语言λProlog。不过，我们的思想相当通用，可用于任何能够正确实现目标中存在量词与全称量词交替序列的逻辑编程环境。[1] 事实上，我们采用的机制在逻辑层面出奇地简单。为支持信息隐藏，我们使用对名称的存在量化。在实践中，名称的隐藏是通过为每个代码模块关联一个签名来实现的；模块中使用但未出现在签名中的所有名称，都被视为隐式存在量化。通过一种称为累积的机制来组合代码单元，对应于公式合取。这会产生一个静态确定的代码空间，但其中谓词定义的可用性可以通过作用域恰当的存在量词加以控制。我们表明，这些简单机制足以实现谓词定义的作用域、数据抽象和模块参数化等特性；以往人们认为，这些特性需要更复杂的动态代码结构化能力。 [7,10]值得注意的是，高阶编程中常见的难题与这种模块化方法完全自然地共存。从实现角度看，累积可以通过编译时内联代码来处理。 [13]然而，真正的模块化需要支持分离编译。我们的第二项贡献在于证明了这一点可以实现。具体而言，我们描述了一种方案，允许分别编译每个模块，并将内联操作推迟到之后的链接时处理阶段。虽然本文没有明确展示这一点，但这一两阶段过程所耗费的总工作量与编译时内联方法相近，并且生成相同的可执行映像。

本文其余部分安排如下。下一节将介绍一种逻辑语言，它包含了刻画我们对模块化的处理所需的所有特性。第3节将描述模块语言的主要组成部分，并说明如何将其翻译为逻辑核心语言。接下来一节将讨论使用范式，以及对基础模块语言的一些实用扩展。第5节将探讨分离编译问题。第6节将通过对比本文内容与相关工作来作结。

[1] 如后文所述，通过改变合一计算，可以为常见的逻辑编程语言添加这样的能力。

2 底层逻辑语言

我们采用存在量化来实现信息隐藏。由于我们希望不仅能够隐藏（第一阶）常量的名称，还能够隐藏谓词和函数的名称，因此，高阶逻辑才是阐述我们这些思想的恰当背景。我们还假定语言是带类型的，尽管我们的主要思想同样适用于无类型的情形。

我们使用一组类型，它最初包含 int、real、string 和 o；此外，还包括使用一元类型构造子 list 构造出的类型，以及最终由这些类型构成的所有函数类型，写作 $\alpha \to \beta$。类型 o 是命题的类型。类型变量用以大写字母开头的词元表示，是无限多个实例类型的简写。语言中的项由带类型的常量和变量符号集合通过应用和抽象操作构造而成。初始常量集包括逻辑符号 true，表示永真命题；$\supset$、$\vee$ 和 $\wedge$，分别表示蕴含、析取和合取的中缀形式；以及两个类型为 ($A \to o$) $\to o$ 的“模式”常量 sigma 和 pi。后两个（族）常量表示广义存在量词和全称量词：$\exists x\, P(x)$ 和 $\forall x\, P(x)$ 在这种逻辑中分别写作 ($sigma\ \lambda x P(x)$) 和 ($pi\ \lambda x P(x)$)。[2] 我们还假定有通常用于表示整数、实数和字符串的常量，以及类型为 list A 的模式常量 nil，还有中缀符号 ::，其类型为 $A \to (list\ A) \to (list\ A)$，它们提供了“内置”的列表概念。应用默认采用左结合。这带来了项的柯里化表示法。如果 p 是一个 n 元关系符号，那么表达式 ($p\ t_1\ \ldots\ t_n$) 表示该关系作用于项 t_1、.....、t_n。如果该表达式的“头部” p 是 true 或非逻辑常量，那么它就是一个原子公式。

在任何实际应用中，都有必要引入新的种类和类型构造器，以及新的非逻辑常量。我们将在下一节介绍声明此类符号的机制。变量也必须关联类型。此类类型信息可以通过推理过程补全，语言语法也可以允许显式提供这些信息。

为了阐述我们的观点，我们将使用一种略作扩展的高阶 Horn 子句理论。这种逻辑的核心组成部分是由以下语法规则定义的 G 和 E 公式：

$$
\begin{array}{lll}
G & ::= & true \mid A \mid G \vee G \mid G \wedge G \mid \exists x\, G \\
D & ::= & A_r \mid G \supset A_r \mid \forall x\, D \mid D \wedge D \\
E & ::= & D \mid E \wedge E \mid \exists x\, E
\end{array}
$$

A 表示此处的原子公式，A_r 表示头部要么是不同于 true 的常量、要么在包围它的上下文中经过存在量化的原子公式。形如 A_r 或 $G \supset A_r$ 的 D 公式称为其头部为 A_r，其体根据具体情况可以为空，也可以是 G。像往常一样，这类公式属于一个谓词的定义，该谓词的名称就是 A_r 公式的头部；这种解释是有意义的，因为接下来描述的计算模型保证该头部最终始终是一个非逻辑

[2] 我们用 $P(x)$ 表示一个可能包含 x 自由出现的项。

常量。限定在一阶语境中，我们的 D 公式本质上是确定 Horn 子句或程序子句的带类型版本，而后者构成了 Prolog 的基础。

核心语言中的计算围绕着尝试解决一个目标展开：该目标由一个 G 公式给出，而一组闭合的 E 公式则定义了一个程序。从语义上说，这一概念通过直觉主义逻辑中的可证性来解释。在过程层面上，这意味着要执行以下步骤。[3] 首先，根据程序和目标中出现的（全局）常量确定一个签名，并用它来确定目标中自由变量的实例化。然后，用新常量实例化 E 公式开头的存在量词，将这些常量加入可用签名，并消去合取，将程序转换为一组形式为 A_r 或 $G \supset A_r$ 的 D 公式。接下来，根据复合目标的结构递归地对其进行化简：合取目标会尝试从同一程序中解决每个合取项；析取目标会尝试解决其中一个析取项；存在量化目标则会根据当前签名，使用一个不包含 pi 或 $\supset$ 的闭项进行实例化。最终，遇到的目标必须是原子的。若该目标为真，则可直接解决；否则，将其与程序中某个 D 公式的闭实例的头部进行匹配：如果体为空，则得到一个解；否则，将该 D 公式的体作为新目标，以相同方式继续处理。

上述计算框架具有一种有趣的处理作用域的能力。具体而言，程序中的存在量化变量对应于名称或常量，而显式量化使我们能够为此类名称指定作用域。因此，在包含公式 $\exists x\, D_1(x)$ 和 $\exists x\, D_2(x)$ 的程序中，出现在这两个上下文中的“名称” x 应被视为不同的常量。这种情况与程序包含公式 $D_1(x)$ 和 $D_2(x)$ 的情况形成对比，假设这两个 x 符合被视为常量的语法标准。存在量化还会影响名称在外部上下文中的可用性。因此，尝试从包含公式 $D_1(x)$ 的程序中求解目标 $G(y)$ 时，允许将常量 x 实例化为自由变量 y 。然而，如果将程序公式改为 $\exists x\, D_1(x)$，则不能这样使用该常量。

3 模块语言及其逻辑解释

在开发实际程序时，首先必须确定一套类型和项常量构成的词汇表。在我们的语言中，新类型构造子通过如下形式的种类声明来定义：

```
kind  tyc1, ..., tycn  type -> ... -> type.
```

其中，构造子 tyc1、...、tycn 的元数比该声明中 type 的出现次数少一。项常量通过如下形式的类型声明来标识：

[3] 这段描述应被视为一种 *precise* 但 *abbreviated* 的表述，这是由于篇幅有限，不得不对通过迁移系统进行的另一种表述加以简化。

类型 c1,...,cn <类型表达式> .

其中，类型表达式由可用的类型构造子构成。

种类声明和类型声明的作用域概念在构造类型和谓词定义时非常重要。模块为这一名称空间赋予了结构。形式上，模块以如下形式的声明开头

module <name>.

随后是与指定名称相关联的种类、类型和谓词定义。这里我们采用面向文件的模块视角：所有用于定义名为 foo 的模块的代码，都存放在名为 foo.mod 的文件中。模块的边界决定了文本层面的作用域概念，即其中出现的种类和类型声明被视为适用于模块中包含的所有其他声明。与这一观点一致，这些边界也为编译器在转换源语言程序的过程中可能执行的分析划定了范围。

下面这个名为 store 的模块定义展示了模块语法：

```
模块存储。 kind 存储 type ->类型。 type emp (存储A)。
type stk A -> (存储 A) -> (存储A)。 type init (存储A) -> o。
type add, remove A -> ( 存储 A) -> ( 存储 A) -> o。 init
emp。 add X S (stk X S)。 remove X (stk X S) S。
```

该模块为存储定义了一种表示，并附带三个操作：一个用于初始化存储，另外两个分别用于向现有存储中添加元素和从中移除元素。为了提供一个与所存储元素的类型无关的定义，该模块声明了一个一元类型构造子来表示存储，并在实现存储表示的常量类型中使用它。此代码中采用的存储具体实现基于栈的概念。模块最后三行是*D* 公式，它们基于这种解释定义了所需的操作。通常，以大写字母开头的记号表示变量，这些变量在公式头部隐式地进行全称量化。

通常，模块中的声明并非都打算对外可见。例如，对于模块存储，隐藏存储的实际表示形式是合理的，这样就必须通过谓词 init、add 和 remove 以不透明的方式操作存储。在我们的语言中，要对外可见的模块内容必须通过一个与模块同名的签名明确指定。形式上，签名以如下形式的声明开头：

sig <name>.

并继续给出要与指定名称相关联的 kind 和类型声明。因此，以下声明构成一个签名，它为模块存储规定了所需的视图类型：

sig store. kind 存储 类型 ->类型. 类型 init (存储A) ->
o. 类型 add A -> (存储 A) -> (存储A) -> o. 类型
remove A -> (存储A) -> (存储A) -> o.

签名有一项显而易见的一致性要求：它们必须标识出合理解释其中所出现类型所需的所有类型构造子。语法上良构的签名实际上充当模块的接口定义：编译器必须将其视为该模块在任何使用场景中所提供的全部类型和种类声明的完整描述，并相应地检查这些声明是否与模块中实际存在的内容一致。为了支持这种类型检查功能，我们将采用一种面向文件的签名视图，与模块的处理方式类似：名为 foo 的签名定义位于名为 foo.sig 的文件中。

到目前为止，我们已经考察了模块和签名的静态语义。动态语义则通过翻译为一个 E 公式来说明，随后使用核心语言的计算模型。这个翻译实际上相当简单：整个模块都可以看作一个（或许很大的）E 公式，它由将其中包含的所有 D 公式合取而成，然后再对该公式中的所有局部常量——即签名中未提及的常量——进行存在量化。采用这种方法，模块存储的逻辑本质，例如，就简化为如下公式

$\exists Emp \exists Stk((init\ Emp)\ \wedge \forall X \forall S(add\ X\ S\ ($
$Stk\ X\ S)) \wedge \forall X \forall S(remove\ X\ (Stk\ X\ S)\ S))$.

模块使用最简单的形式发生在交互层面。特别是，在顶层尝试求解目标时，总是相对于所选模块进行。与前文讨论一致，从词汇角度看，这种相对化的意义在于，模块签名中的所有种类和类型声明，在分析目标的语法时都可用。另一方面，这种相对化在计算上的意义是，求解目标时只使用模块中全局谓词的定义，并且最多只能访问其全局常量。这种解释实际上意味着一种强形式的隐藏。具体来说，假设我们正尝试从模块 M 求解目标 G 。显然， G不能直接引用任何属于 M 的局部常量。稍微不那么明显的是，这些局部常量也不能以计算结果的形式从 M 中泄露出来。因此，假设 G 是一个唯一自由变量为 y 的目标。那么，此查询计算出的结果就是使 G 成功求解的 y 的实例化。模块的翻译语义以及

相关的操作语义现在明确排除了 M 的局部常量出现在 y 的实例化中的可能性。[4]

模块系统的一个重要方面，是它提供了哪些机制来实现代码单元之间的交互。在我们的系统中，这种能力通过一个称为模块累积的单一操作实现。通过此操作组合模块的方法，是在正在构建的新模块中放置如下形式的声明：

累积 M1, ..., Mk。

假设 M1, ..., Mk 是模块名称。该声明的词法效应是在新模块中提供对模块 M1, ..., Mk 的签名的访问。再次通过诉诸模块的逻辑本质来规定这一构造的动态语义。设 $E_1, \ldots, E_k$ 是与模块 M1, ..., Mk 对应的公式。那么，与新模块对应的公式首先通过将 $E_1, \ldots, E_k$ 与忽略累积部分后得到的模块公式合取而得；随后，将每个 E 公式头部的存在量化提升至整个合取式的作用域之上。当然，移动存在量词时必须保证逻辑正确，例如，$(\exists x D_1(x)) \wedge (\exists x D_2(x))$ 应转换为 $\exists y \exists z (D_1(y) \wedge D_2(z))$，其中 x 中的 D_1 和 D_2 要分别重命名为互不相同且未使用过的变量 y 和 z。在程序层面上，这种逻辑解释相当于将累积视为内联所有被累积模块中的代码，但同时要注意保留各模块中名称的局部性。

当然，一个累积模块本身也可以累积其他模块。在这种情况下，只有当累积链不会回溯到自身时，这一构造的逻辑解释才有意义。因此，语言语法禁止这类循环。[5] 在独立编译模型中，这一约束只有到链接时才能检查。

至此，我们已经完整描述了模块语言的基本结构：从语法上说，一个模块对应一个可能很大的 E公式，该公式按刚才所述的方式，将模块的子句、签名以及与其所累积模块相对应的公式组合而成。随后，通过这个公式以及核心语言的操作语义，就能完整而精确地说明该模块的动态语义。下一节将在不修改逻辑核心的前提下，为该语言增添更多语法糖和一些面向编译器的注释。

4 模块的实际应用

虽然上一节描述的模块语言很简单，但在编程层面却相当灵活。本节将通过探讨它的一些使用范式，展现这一特点。

[4]与语义不同的一个问题，是如何高效处理此类实例化约束。我们曾在别处展示过如何通过为逻辑变量和常量赋予数值标签，并在合一中使用这些标签来实现这一点 [11]。[5] 请注意，累积到同一上下文中的两个模块 *can* 会使用彼此定义的谓词，因此可能相互依赖。因此，禁止累积链中出现循环， *not* 就等同于禁止这类依赖关系。

```
模块 graph_search. 累积存储。kind 状态，动作 类型。type graph_search
list action ->  o. type init_open store state ->o. type expand_graph store
state ->  list state ->list action ->  o. ... graph_search Soln :-init_open
Open, expand_graph Open nil Soln. init_open Open :- start_state State,
init Op, add State Op. expand_graph Open Closed Soln :- remove State
Open Rest, final_state State, soln State Soln. expand_graph Open Closed
Soln :- expand_node State NStates, add_states NStates ROp (
State::Closed) NOp, expand_graph NOp (State::Closed) Soln. ...
```

图 1. 实现图搜索的模块

我们语言中的模块允许将支持所需功能的代码收集到一个具名单元中，而关联的签名则提供了查看这些代码的窗口。模块实现的功能可能需要在另一个模块的上下文中使用。模块累积支持实现这种交互。例如，假设我们希望实现一种基于启发式的图搜索过程。该过程会初始化一组状态，然后依据生成新状态的规则以及用于选择下一个要扩展状态的底层策略，扩展这组状态。为了实现所需功能，这段代码可能需要存储的实现。可以通过累积前文介绍的模块 store 来获得该实现。图 1 展示了图搜索模块定义的一部分，以说明这一思路。累积 store 后，该模块便拥有一种存储类型，供 init_open 和 expand_graph 的类型声明使用。这种累积还提供了 init、add 和 remove 操作。值得注意的是，尽管该模块程序子句中的（全称量化）变量可以实例化为存储表示，但这些表示仍然是抽象的：对局部常量的隐式存在量化施加了可见性限制，确保只有借助模块 store 中的操作才能访问其内部结构。还请注意，通过不在 graph_search 的签名中声明 init、add 和 remove，store 中的谓词定义便可以完全对图搜索模块私有。因此，与 [7], 的导入构造不同，我们只需对命名空间进行组织，就能在一个完全静态确定的代码集合中实现代码作用域。

上面的示例展示了模块如何获得一份私有的代码副本。在某些情况下，这或许正是所需的行为；但在许多其他情况下，累积而成的模块可能代表一种通用能力，需要在大型系统的不同模块之间共享。一个具体的示例

是列表上的 append 等“库”谓词：鉴于列表无处不在，这些谓词很可能会在许多地方用到，因此最好在整个系统中使用同一份代码，而不是在每个需要它的地方都复制一份。

解决这一问题的一种方法，是将使用此类库功能的模块视为以这些功能为参数。我们的模块语言以一种自然的方式支持这种参数化。例如，假设我们希望将模块 graph_search 视为依赖于外部提供的存储实现，而不是由它自身累积存储。这种依赖关系可以通过在其签名中加入适当的声明来明确体现。具体来说，该签名会标明类型构造子 store，以及常量 init、add 和 remove。当然，我们最终还需要为该模块提供所需的功能。这可以通过在相关位置累积模块 store 来实现。具体来说，假设我们想测试图搜索的实现。可以通过以下模块编写适用于此目的的测试框架：

模块 test_graph_search。
累积 graph_search、store。
通过为该模块赋予一个签名，使模块 graph_search 中所需的类型、数据表示和谓词对外可见，我们便可以针对它提出查询，以检验待测试的功能。[6]

在逻辑编程语境中，可以通过添加更多子句来扩展谓词定义。由某个累积模块带来的定义，也可以在累积模块中以这种方式扩展。通常，我们希望限制这种可能性。例如，参考模块存储，我们可能希望冻结它所提供的操作 init、add 和 remove 的定义。通过允许在签名中使用“export”注解，可以支持这种做法。具体而言，将

type add A -> (store A) -> (store A) -> o. 替换为带注解的声明

exportdef add A -> (存储A) -> (存储A) -> o.
在模块 store 的签名中，我们可以表明，累积上下文不得更改 add 的定义。考虑到我们刚才概述的配对功能模型，我们的模块语言还提供了与之互补的“useonly”注解。因此，使用声明

```
useonly  add  A -> (store A) -> (store A) -> o.
```
在模块图_search 中，我们不写出 add 的类型声明，而是指出该谓词在此模块中只能原样使用。我们注意到，在逻辑层面，exportdef 和 useonly 声明与类型声明完全相同。它们与类型声明的区别仅在于实用层面：它们对模块组合施加了特殊的良构性限制——这些限制必须经过检查，且编译器可以利用这些限制——并且

[6] 这一讨论也说明了复合系统中的不同模块如何共享“库”功能：它们可以按上述方式参数化，而后通过将库模块累积到一个公共上下文中来解除参数。

可用于模块组合。不过，这些限制在实践中相当有用：它们赋予定义一种完备性，有助于推理程序的性质，也有助于生成更优的目标代码，尤其是在独立编译的情境下。

我们所描述的参数化思想要求在多个位置重复写出签名。为了简化程序结构，从而提供更好的文档说明，我们提供了累积签名的功能。要将一个签名包含在另一个签名或模块中，可以使用如下形式的声明

accum_sig foo.。这里有一点需要注意：签名 foo 可能包含对谓词的注释，将其标记为 exportdef 类型，而在累积上下文中，可能需要补充这些谓词。此时可以使用声明

use_sig foo.
来代替。该声明也会原位包含所提及的签名，但前提是先将每个 exportdef 声明转换为 useonlyone。

谓词定义通常完全由特定模块确定，这可能是因为本节所述类型的注释，也可能是因为这些谓词涉及局部常量；但在需要时，谓词定义也可以分布在相互作用的模块中。Prolog 中的多文件声明便证明了这一功能有一定用途。另一个例子是，考虑在不同逻辑中实现证明关系这一任务。所有这些逻辑的共同部分可能是对命题规则的处理。这部分处理可以单独放在一个特定模块中，例如命名为 prop_logic。随后，一阶逻辑的实现可以累积 prop_logic，并扩展其中定义的谓词。虽然跨越模块边界扩展定义的能力很有用，但它也会给分离编译带来特殊问题，我们接下来将对此进行讨论。

5 分离编译方案

我们所描述语言的一种朴素实现，可以通过在编译时内联累积的模块来实现。本节将说明，这种方法可以改进为一种方案：每个模块单独编译，内联则在稍后的链接阶段完成。为揭示需要解决的问题，我们先概述下面的朴素方法。然后，我们以此为背景，提出改进后的分离编译方案。

5.1 模块累积的一种朴素实现

图 2 展示了一个典型示例，说明实现必须能够处理哪些模块交互。在这个示例中，我们没有明确展示签名，而是直接在模块代码中将常量标记为全局或局部。为求简洁，我们还省略了类型声明。

通过内联累积的模块，可以实现我们语言的一种朴素实现。不过，这一过程必须谨慎区分来自不同累积模块的常量，并将它们映射到所构造的更大模块中作用域适当的常量。A

模块 m1。模块 m2。全局 r,w。全局 r。[m 中的子句1] [m 中的子句2]模块 m3。模块 m4。模块 m5。累积 m1。累积 m2。累积 m3,m4。局部 r。局部 r, w。局部 q。全局 w。全局 w。[m 中的子句3] [m 中的子句4] [m 中的子句5]

图 2. 嵌套累积示例

模块 m5. 全局 w[1]. 局部 q, r[1]，w[2] ,r[2]. {累积 m3 [r -> 第二个局部：r[1]，w -> 第一个全局：w[1]]全局 w. 局部 r.[来自 m3]的子句，常量引用已适当解析 {累积 m1 [r -> 第一个局部：r, w -> 第一个全局：w]I来自 m1]的子句，常量引用已适当解析}} {累积 m4 [r -> 第四个局部：r[2]，w -> 第三个局部：w[2]]局部 r,w.[来自 m4]的子句，常量引用已适当解析 {累积 m2 [r -> 第一个局部：rI来自 m2]的子句，常量引用已适当解析}}[来自 m5]的子句，常量引用已适当解析

图 3. 内联累积后模块 m5 的定义

这种内联编译器必须完成的工作的示意图见图 3。这里我们使用名称 r[1]和 r[2]来区分来自模块 m3 和 m4、名称均为 r 的两个常量；类似地，我们使用名称 w[1] 和 w[2] 来区分模块 m5 中的全局常量与模块 m4 中的局部常量，它们的名称均为 w。此外，我们将重命名过程展开为沿累积链的一系列步骤。例如，模块 m1 子句中的常量 r 要重命名为封闭上下文（模块 m3）中的第一个局部常量，而该局部常量本身又要重命名为最外层上下文中的第二个局部常量。实际上，内联编译器可以通过实际执行这一系列重命名操作，将这种嵌套关系折叠起来，从而得到一个包含一组全局和局部常量的模块，以及来自所有模块的一组子句，其中的常量均已得到恰当标识。随后，编译器便可以在全面掌握所有相关谓词定义的基础上编译这些子句。

5.2 一种基于分离编译的处理方法

前一种模型指出了执行前需要生成的代码结构。然而，我们感兴趣的是从

各个模块 m5、m4、m3、m2 和 m1 的已编译版本生成这段代码。这些版本是在不知道它们将在哪里使用、且至多只掌握其所累积模块的签名信息的情况下生成的。在这种情况下

1. 编译器在编译一个可能会被累积的模块时，不会明确知道全局和局部常量在外围上下文中将被映射到什么；2. 对于名称为全局且定义可扩展的谓词，编译器必须生成代码，并假定模块中的子句构成一个不完整集合，必须将其纳入更大的上下文；3. 编译器生成的代码可能包含对某些谓词的调用，而这些谓词的入口点在编译时无法确定，必须等到相关模块组合完成后才能确定。

在这种情况下，内联编译器的任务必须由两部分分担：一个编译器分别为每个模块生成代码，并在代码中加入适当的注释，以便将其纳入更大的上下文；另一个是链接器，它利用每个模块附带的“粘合”信息，构建系统完整的字节码映像。下面我们将简要介绍这些组件的结构，它们协同工作即可实现所需结果。为便于阐述，我们假设目标低级代码可在一种与 Warren 抽象机（WAM） [17] 密切相关的体系结构上运行。以下内容假定读者熟悉这种机器结构。

编译的结果 给定一个模块，我们设想的编译器会生成一个供链接器使用的文件，其中包含以下信息：

1. 全局常量列表，其中包含常量名称以及执行期间所需的其他信息，例如类型。2. 局部常量列表，与全局常量列表类似，但这次不需要名称。 3. 每个累积模块的名称列表，并将其全局名称（从其签名中获得）映射到该模块的局部常量列表或全局常量列表中的索引。 4. 可外部重定义的谓词（索引）列表。 5. 编译模块中所列子句后生成的类 WAM 代码。代码中的常量索引指向头部中的列表，最终由链接器修补。对可外部重定义谓词的调用也使用这些谓词列表中的索引，必须等到最终确定这些谓词的入口点后才能填入。 6. 谓词名称（以索引表示）到其在代码空间中的入口点的映射。

在 WAM 环境中，为定义某个谓词的各个子句生成的代码，前后都配有用于依次尝试各个选择以及对其进行索引的指令。其典型结构如下所示：

```
        try_我_else L1 switch_对_项
V1,C1,Lst1,S1 C1: switch_对_常量 CHT1
```

S1: switch_对_结构SHT1 V1: try_
我_else L12[单个子句的代码]
L12: retry_我_else L13 ... LLn:
trust_我[最后一个子句的代码]
L1: retry_我_else L2[另一个代码
块的代码]... Ln: trust_我[最后一个
代码块的代码]

在最外层，这段代码捕捉了在不同子句块之间可能存在的顺序关系。每个块对应一个子序列，对该序列进行索引可能会有所帮助。“已索引”子序列的一个组成部分涉及变量情况，必须支持对整个集合进行简单的顺序处理。另一种可能性对应于列表，或对常量名称或结构名进行哈希；在这种情况下，子列表中只有部分子句是相关的。此时会生成使用 try、retry 和 trust 指令的辅助顺序代码。上文并未具体展示这些可能性，但它们与接下来的讨论仍然相关。

链接过程链接首先会为顶层模块的扁平化映像创建一个帧，随后由一些函数填充该帧：这些函数将全局和局部常量映射到运行时索引，递归加载累积的模块，最后将代码添加到帧中。

将常量映射到运行时索引的过程遵循内联编译器的级联结构，只不过这一步是在链接时完成的。具体来说，链中的每个局部常量都会映射为一个唯一索引。顶层模块的全局常量也会被分配唯一索引。最后，父模块的局部和全局常量的分配，以及相关的重命名函数，共同决定了累积模块中全局常量的索引。

在准备将代码添加到帧中时，一个重要问题是如何合并谓词定义：不同模块可能分别提供某个谓词定义的一部分，因此需要将这些部分组合起来。首先，组合时必须遵循固定顺序——例如，在 λProlog 中，来自累积模块的子句按累积顺序排列在前，随后是父模块中的子句——链接器会遵循这一顺序。现在，每个模块中为谓词生成的代码都具有列表结构，整合时自然的第一步就是将各个列表 append 在一起。暂且不考虑 WAM代码中索引的存在，很容易看出如何做到这一点。例如，假设需要合并下面显示的左右两部分中所示形式的两段已编译子句序列：

L1：try_me_else L2 L4 try_me_else L5[代码块代码
]　　[代码块代码]L2：retry_me_elseL3 L5：
retry_me_elseL6[代码块代码]　　[代码块代码]
L3：trust_me L6：trust_me[代码块代码]　　[代
码块代码]

可以通过将第一组列表最后一个块之前的 trust_me 指令改为指向下一个定义起始位置的 retry_me_else 指令，并将该集合的第一条指令改为 retry_me_else，来实现这种组合，结果如下：

L1: try_me_else L2 L4 retry_me_else L5[块的代码
]　　[块的代码]L2: retry_me_else L3 L5: retry_
me_else L6[块的代码]　　[块的代码]L3: retry_
me_else L4 L6: trust_me[块的代码]　　[块的代码]

WAM 中的索引优化使情况稍微复杂了一些，因为顶层序列中的某些元素可能是索引块。如果像前面建议的那样直接 append 顶层序列，新序列中可能会出现两个相邻的此类块。这并不可取：例如，这可能导致我们在实际上不需要时仍将一个选择点保留在栈上。幸运的是，可以避免这种情况。我们可以检查第一个序列的最后一个元素和第二个序列的第一个元素，来判断是否会发生这种情况。如果它们都是索引块，那么我们就将它们合并。

生成单个索引块时需要解决的一个问题，再一次涉及合并两个表示按子句顺序执行的代码段。对于与变量情况对应的“主”序列，可以像处理顶层序列那样处理。为了处理这些序列可能通过 try、retry 和 trust 指令实现的情况，我们在指令集中增加了两条新指令，称为 try_else 和 retry_else。这些指令的行为类似于 try 和 retry，只是它们还接受一个额外参数，用于提供回溯时要尝试的代码地址。现在假设需要合并的两个顺序执行代码块如下：

S1：try L1	S2：try L4
retry L2	retry L5
trust L3	trust L6

可以通过将代码改为以下形式来实现这种合并：

S1: try L1	S2: retry L4
retry L2	retry L5
retry_else L3，S2	trust L6

在第一个代码块对应唯一子句选择的情况下，实现这一思路需要用到 try_else 指令。

合并索引代码块时需要解决的另一个问题，是如何合并与常量名和结构名对应的哈希表。这很容易做到。编译器实际上可以将各个独立的表直接输出为由常量名称及其对应代码入口点组成的列表。链接器可以据此确定需要合并哪些列表，然后输出合并后的列表，模拟器再利用这些列表生成哈希表。

链接器必须解决的最后一个方面，是那些无法在编译时确定的谓词的（相对）代码位置。所有子句代码合并完成后，即可获得一张映射表，将通用命名空间中的每个谓词名称映射到其定义的位置。现在可以在相应位置填入这些地址。

6 相关工作与结论

本文阐述了逻辑编程中模块化的一种基于逻辑的解释。评估本文贡献时自然会提出一个问题：本文提出的处理方式与函数式编程中的处理方式有何关系？在实践层面，本文的思路与 Standard ML 中的签名和结构，以及程序中的存在量化，显然有相似之处；后者看起来与函数式编程中隐藏机制所依据的存在类型颇为相似 [9]。然而，两者也存在根本差异：逻辑编程基于证明搜索，而非证明正规化。本文着重阐明逻辑编程模块化的一种连贯的逻辑观点，并进一步说明如何为相关的证明搜索提供令人满意的计算处理方式。

另一项相关比较是与赋予实际 Prolog 系统模块化能力的研究。此类工作（例如 [3,15,16]）的一个主要关注点，是如何解释 call 等元逻辑谓词，以及如何处理与语法有关的声明。对这种模块化观点的关注具有重要的实践意义——例如，参见 [3] ，了解其对语言可扩展性的重要性。然而，这一关注点与我们在此探讨的名称与代码作用域问题并不相干。就名称处理而言，其他方法在逻辑层面上多少有些临时性，只允许隐藏谓词名称，却不允许隐藏函子和常量的名称，而这正是数据抽象的关键方面。Mercury 语言中的处理方式在实践上与我们的方法更为接近，但有所不同：它要求隐藏某类型的所有构造子，或者一个也不隐藏 [1]。此外，我们对存在量词的使用能够带来更丰富的计算，这需要更复杂的合一过程。遗憾的是，篇幅所限，无法对此问题作更全面的讨论。

我们还应将本工作与逻辑编程领域中侧重于以逻辑为中心的方法来实现模块化的研究进行比较。Sanella 和 Wallen 的提案 [14] 将 ML 的思想引入逻辑编程环境，是其中一个例子。该提案中的签名和结构概念，与我们对签名和模块的理解再次十分接近。一个区别是， [14] 不允许在不同结构/模块之间逐步构建谓词定义；正如第4节所论证的，这种能力在逻辑

编程环境中可能很有用。我们还注意到，在我们的环境中，通过签名规范实现的隐藏是以基于逻辑的方式来解释的。另一个提案来自 Miller [8]，它涵盖了我们在此使用的各种构造。相较于这项工作，我们的贡献在于证明，其完全静态的一个子部分也足以实现对子句定义的作用域控制。最后，我们提及 Harper 和 Pfenning [6] 的工作：他们将类 ML 方法的模块化方案应用于基于 LF 的逻辑编程语言，但与 [7], 一样，也允许动态修改谓词定义。

在实现层面，我们必须处理两个不同的问题：存在量化变量的作用域处理，以及如何合并由不同模块编译得到的同一谓词的代码。第二个问题也与例如 SICStus 系统中的多文件定义有关。该系统采用的解决方案至少表面上有所不同：编译直接面向核心语言进行，索引则通过解释方式实现，其依据是在编译每个子句时逐步构建的数据结构 [4]。不过，值得探讨的是，这种方法的某些方面是否也能适用于我们的分离编译设置。

与本文工作相关的方面有很多，例如支撑我们模块化方案的逻辑特性、为实现这一概念而选择的语法、编译与计算（或者更准确地说，合一）中的名称作用域处理，以及分离编译的实现。这些方面过去都曾分别得到研究，因此引出了一个问题：本文的具体贡献究竟是什么？我们认为，本文的主要创新在于将这些不同的思想结合起来，提出一种以逻辑为出发点的模块化方法；这种方法在实践中有用，并且其实现简单，基于分离编译。

我们所描述的模块语言已在 Teyjus 系统中实现。对于我们所倡导的作用域处理方法，相关使用经验是积极的：用户已轻松地从同样受支持的基于动态导入的方法转而采用这种方法。第5节提出的理念是该系统重新实现工作的一部分；新版本即将发布，并且现在也支持分离编译。

7 致谢

本研究得到美国国家科学基金会资助，资助编号 0429572。本文所表达的观点、研究结果、结论或建议均属作者所有，不一定反映美国国家科学基金会的意见。

参考文献

1. Ralph Becket。《Mercury 教程》，2005年。可通过网址 http://www.cs.mu.oz.au/research/mercury/tutorial/book/book.pdf 获取。2. F. Bueno、D. Cabeza、M. Carro、M. Hermenegildo、P. L ´ *The* opez *Ciao Prolog System* 1997 3 97, 和 G. Puebla。，August 。参考手册，技术报告 CLIP / School of Computer Science，马德里理工大学。

3. D. Cabeza 和 M. Hermenegildo。Prolog 的新模块系统。载于 *Computational Logic - CL 2000* 131 148 2000 1861 ，第 – 页。

Springer， 。人工智能讲义第 卷。4. M. Carlsson。私人通信，2007年6月。5. 国际标准化组织。Prolog。ISO/IEC 13211 — 第 2 部分：模块，2000年。6. R. Harper 和 F. Pfenning。基于 LF逻辑框架的编程语言模块系统。 *Journal of Logic and Computation*，8(1):5–31，1998年。7. D. Miller。逻辑编程中模块的逻辑分析。 *Journal of Logic Programming* 6 79 108, 1989 ，：–。8. D. Miller。 λ *Proceedings of the 1993 Workshop on Extensions to Logic Programming* 206 221 Prolog 模块提案。载于 R. Dyckhoff 编，《》，第 – 页。Springer-Verlag，1994年。《计算机科学讲义》第 798 卷。9. J.C. Mitchell 和 G.D. Plotkin。抽象类型具有存在类型。 *ACM Trans.*，( ):–。10. *Program. Lang. Syst.* 10 3 470 502, 1988
L. Monteiro 和 A. Porto。上下文逻辑程序设计。载于 G. Levi 和 M. Martelli 编，《 *Sixth International Logic Programming Conference*》，第 284–299 页。麻省理工学院出版社，1989年6月。11. G. Nadathur。遗传Harrop公式逻辑的证明过程。 *Journal of Automated Reasoning*，11(1):115–145，1993年8月。12. G. Nadathur 和 D.J. Mitchell。系统说明：Teyjus——

λ *Au-* Prolog 的编 *tomated Deduction–CADE-16* 1632 译器和基于抽象机的实现。载于 H. Ganzinger 编，《》，《人工智能讲义》第 号，第 287–291 页。Springer-Verlag，1999年7月。13. G. Nadathur 和 G. Tong。在

λ *Journal of Functional* Prolog中实现 *and Logic Programming* 1999 9 1999 模块化。，( )，April 。14. D.T. Sannella 和 L.A. Wallen。构造模块化Prolog程序的演算。 *Journal of Logic Programming*，12:147–178，1992年1月。15. 瑞典计算机科学研究所。 *SICStus Prolog v3 User's Manual*。智能系统实验室，PO Box 1263，S-164 28 基斯塔，瑞典，1991–2004年。16. 瑞典计算机科学研究所。 *Quintus Prolog v3 User's Manual*。智能系统实验室，PO Box 1263，S-164 28 基斯塔，瑞典，2003年。17. D.H.D. Warren。抽象Prolog指令集。技术说明 309，SRI International，1983年10月。