arXiv:2302.00558v1 [cs.PL] 1 Feb 2023

受 Oz 启发的 Prolog 未来发展构想

Peter Van Roy 鲁汶天主教大学

Seif Haridi 瑞典皇家理工学院

2022年9月16日

摘要

Prolog 和 Oz 都是以逻辑编程为核心的多范式语言。Oz 中有一个重要子集在语法上是 Prolog 的变体：带有绿色或蓝色剪枝以及 bagof/3 或 setof/3 的纯 Prolog 程序可以直接转换为 Oz。鉴于 Prolog 与 Oz 之间的密切关系，我们认为 Oz 对逻辑编程所作的扩展可以为 Prolog 未来的发展提供启发。我们介绍三种扩展，即确定性逻辑编程、惰性并发函数式编程和纯函数式分布式计算。我们简要介绍这些扩展，并说明它们如何帮助 Prolog 在未来 50 年不断发展。

1 引言

Prolog 语言诞生于 50 年前，如今到了 2022年，仍被广泛使用。人们过去和现在都在为 Prolog 定义各种可能的扩展，使其能够作为一种不断发展的语言延续下去。Oz 语言从 1991年开始开发，并于 1999年推出了一个重要的系统版本，名为 Mozart [13]。Oz 的主要目标是以简洁的方式支持多种编程范式。Oz 语言和 Mozart 系统曾广泛使用了约十年，但之后使用率有所下降。2009 年之后直到今天，它主要仅用于编程教学。这与 Prolog 不同，Prolog 至今仍广泛使用。

尽管使用率有所下降，Oz 率先提出了许多重要概念，而自其发布以来，这些概念已逐渐成为主流。我们所说的包括确定性数据流计算、返回未来值的角色、通过计算空间对约束编程的深度支持，以及对分布式计算的深度嵌入。这一点与 Prolog 密切相关，因为 Oz 与 Prolog 之间存在紧密的语义关系。Oz 的语义基础是并发约束编程，这是对 Prolog 执行模型的泛化 [7]。具有逻辑语义的 Prolog 程序可以直接转换为 Oz，具体说明如下

第2节。鉴于这些事实，我们相信，Oz 的设计可以为 Prolog 的未来演进提供启发，使 Prolog 能够再取得 50 年的成功。本文旨在介绍 Oz 对 Prolog 风格逻辑编程所作出的三项重要扩展，并解释为什么它们能够为 Prolog 的未来发展提供启发：

- 确定性逻辑编程（第3节）。我们展示 Oz 如何支持编写逻辑程序，使其能够管理确定性执行，同时保持其逻辑语义。对于给定的逻辑语义，我们展示 Oz 如何表达不同的操作语义。

- 惰性并发函数式编程（第4节）。我们展示 Oz 的逻辑编程核心如何成为一系列函数式编程范式的基础，并最终发展为惰性确定性数据流。所有这些范式都保留了纯函数式编程强大的合流性，同时也延续了它们源自确定性逻辑编程这一特点。

- 纯函数式分布式计算（第5节）。Oz 社区采用深度嵌入方法构建了 Oz 的分布式实现，其中每个语言实体都有自己的分布式算法。作为这项工作的一部分，我们设计并证明了分布式有理树合一的正确性。基于使用逻辑变量进行数据流同步，我们展示了异步消息传递可以是纯函数式的。最后，我们指出，当今使用的大多数实用分布式系统过度依赖非确定性；更好的做法是将它们设计为以函数式为主，仅在少数确有需要的地方加入非确定性。

这些扩展各自在单独的章节中介绍。对于每种扩展，我们都会阐述其主要思想，并提供参考资料，供希望进一步探究的读者查阅。作为准备工作，第2节将说明如何把 Prolog 程序转换为 Oz 程序，以证明 Oz 是 Prolog 的一种语法变体。

2 Oz 是 Prolog 的一种语法变体

我们将说明如何把任何扩展了绿色或蓝色剪枝以及 bagof/3 或 setof/3 谓词的纯 Prolog 程序，转换为具有相同逻辑和操作语义的 Oz 程序。我们假设读者对 Prolog 有足够的了解，能够理解我们的示例。为求简洁，我们不详细解释 Oz 语句的语义；从示例中应该可以清楚地看出它们的含义。关于 Oz 与 Prolog 之间关系的更完整、正式的解释，请参阅 [12] 的第9章。

Oz 和 Prolog 都支持通过合一来绑定的符号数据结构。与许多现代 Prolog 系统类似，Oz 使用有理树合一。Oz 与

Prolog 数据结构在语法和语义上略有不同，主要是因为 Oz 支持记录等额外的数据结构。这里不解释这些差异；关于 Oz 数据结构的精确定义，请参阅 [12] 。

2.1 确定性谓词

如果一个谓词的所有子句在逻辑上互不相交，那么它就是确定性的。当这些子句具有互不相交的守卫条件时，便属于这种情况。确定性谓词使用 Oz 语句 **`if`** 和 **`case`**进行翻译，这两者都具有逻辑语义。来看下面这个确定性 Prolog 谓词：

```
place_皇后(I,_，_，_) :- I==0, !. place_皇后
(I, Cs, Us, [_|Ds]) :- I>0, J is I-1, place_皇
后(J,Cs, [_|Us]，Ds), place_皇后(I, Cs,
Us, Ds).
```

根据 O'Keefe 的说法，这个谓词包含蓝色剪枝 [6], ，也就是说，剪枝是必要的，它能让朴素实现得知该谓词是确定性的，从而提高效率，但不会改变程序的结果。谓词 place_queens/4 可翻译为以下 Oz 代码：

```
过程 {PlaceQueens N Cs Us Ds}如果 N==0
那么跳过 否则如果 N>0 那么 Ds2 Us2=_|
Us在Ds=_|Ds2 {PlaceQueensN-1 Cs Us2
Ds2} {PlaceQueen N Cs Us Ds}否则失败 结束
结束
```

2.2 非确定性谓词

当一个谓词的子句在逻辑上并不互斥时，它就是非确定性的。此类谓词可以进行回溯，从而可能得到多个解。非确定性谓词使用 Oz 语句 **`choice`** 进行转换。请看以下非确定性 Prolog 谓词：

```
place_queen(N, [N|_], [N|_], [N|_]). place_
queen(N, [_|Cs2], [_|Us2], [_|Ds2]) :- place_q
ueen(N, Cs2, Us2, Ds2).
```

该谓词被转换为一个非确定性的 Oz 过程：

```
过程 {PlaceQueen N Cs Us Ds}choiceN|_ =Cs
N|_ =Us N|_ =Ds[] _|Cs2=Cs _|Us2=
Us _|Ds2=Ds在{PlaceQueenN Cs2 Us2 Ds2}结束
结束
```

2.3 一等 Prolog 顶层

前面两个谓词是一个程序的一部分，该程序用于求解 n 皇后问题：如何在 $n \times n$ 棋盘上放置 n 个皇后，使它们彼此都无法攻击。为完整起见，我们在此给出完整程序：

```
函数{Queens N}函数 {MakeList N}如果 N==0 那么 nil 否则 _|
{MakeList N-1} 结束 结束Qs={MakeList N}过程 {PlaceQueensN Cs
Us Ds} /* 如前 */结束 过程 {PlaceQueen N Cs Us Ds} /* 如前 */结束
于{PlaceQueensN Qs _ _} Qs结束
```

这个程序通过调用系统过程 `SolveOne` 来运行，该过程对应于一个 Prolog 顶层查询：

```
{浏览 {SolveOne 函数 {$} {皇后 8} 结束}}
```

语法 **fun** `{$} ... ` **end** 定义了一个零参数函数，它调用单参数函数 `Queens`。系统过程 `Browse` 会显示其参数。`Browse` 显示的答案是一个列表，其中给出了第一个解：

```
[[1 7 5 8 2 4 6 3]]
```

Oz 还提供了 `SolveAll` 来计算所有解，以及 `Solve` 来计算解的惰性列表。`Solve` 与 Prolog 顶层十分相似，解会按需计算。

2.4 带绿色或蓝色剪枝的谓词

如果你的 Prolog 程序使用剪枝“!”，那么当该剪枝属于 O’Keefe [6] 所定义的绿色剪枝或蓝色剪枝时，将其翻译为 Oz 就很简单。绿色剪枝会移除无关的解。蓝色剪枝则向编译器表明该程序是确定性的。为了展示这种翻译方案，我们来翻译下面这个谓词：

```
foo(X, Z) :- guard1(X, Y), !, body1(Y, Z). foo(X, Z) :-
guard2(X, Y), !, body2(Y, Z).
```

这两个守卫谓词不得绑定任何头部变量，也就是说，它们是静默守卫。在 Prolog 中，良好的编程风格是在剪枝之后再绑定头部变量。根据守卫条件是否具有确定性，翻译分为两种情况。如果某个守卫条件具有确定性（没有**choice**），就可以将其写成一个确定性布尔函数。这样便可得到以下简单的翻译：

```
过程{Foo X Z}若 Y 属于 {Guard1 X Y} 则 {Body1 Y Z}否则
若Y 属于 {Guard2 X Y} 则 {Body2 Y Z}
```

否则失败结束
end

如果守卫谓词是非确定性的（它使用 **choice**），那么可以将其写成带有一个输入参数和一个输出参数的形式，如下所示：`{Guard1 In Out}`。它不应绑定输入参数。这就得到以下翻译：

过程{Foo X Z}**情况** {SolveOne **函数** {$} {Guard1 X} **结束**} **为** ［Y］ **那么**
{Body1 Y Z}**否则情况** {SolveOne **函数** {$} {Guard2X} **结束**} **为** ［Y］ **那么**
{Body2 Y Z}**否则 失败 然后结束**

如果这两种情况都不适用于你的 Prolog 程序，例如，你的守卫条件绑定了头部变量，或者你以其他方式使用剪枝（即作为红色剪枝），那么你的程序很可能没有逻辑语义。带有红色剪枝的程序仅由其操作语义定义，这超出了我们的翻译方案的作用域。

2.5 `bagof/3` and `setof/3` 谓词

Prolog 的 bagof/3 谓词相当于在 Oz 程序中使用 `SolveAll` 。它的扩展形式 setof/3 会对结果排序并移除重复项。这可以通过 Oz 内置的 `Sort` 操作完成。我们将展示如何在不使用和使用存在量化的情况下翻译 bagof/3。请看下面这个小型圣经数据库（灵感来自 [10]）：

父亲关系(terach, abraham). 父亲
关系(abraham, isaac). 父亲关系
(haran, milcah). 父亲关系
(haran, yiscah).

在 Oz 中可以写成如下形式：

proc {Father F C}**choice**F=他拉 C=
亚伯拉罕[] F=亚伯拉罕C=以撒
[] F=哈兰 C=密迦[] F=哈
兰 C=亦迦**endend**

调用 bagof/3 时不进行存在量化，例如：

children1(X, Kids) :- bagof(K, 父亲关系(X,K), Kids).

使用 `SolveAll` 定义如下：

过程{Children1X Kids} {SolveAll **过程**{$ K} {父亲关系 X
K} **结束**Kids}**结束**

`Children1`定义具有确定性；它假设`X` 已知，并返回`Kids`。要搜索`X` 的不同值，则应改用以下定义：

```
过程 {Children1X Kids} {父亲关系 X _} {SolveAll 过程 {$
K} {父亲关系 X K} 结束 Kids}结束
```

调用 `{Father X _}` 会在 `X` 上创建一个选择点。“_”是 **`local`** `X` **`in`** `X` **`end`** 的语法糖，它只是一个作用域非常小的新变量。使用存在量化调用 bagof/3，例如：

```
children2(Kids) :- bagof(K, X^父亲关系(X,K), Kids).
```

定义如下，使用 `SolveAll`：

```
过程 {Children2 ?Kids} {SolveAll 过程 {$ K} {父亲关系_
K} 结束 Kids}结束
```

Oz 方案使用 _ 来添加一个新的存在作用域变量。相比之下，Prolog 方案引入了一个新的语法概念，即“存在量词” X^，它只在 setof/3 和 bagof/3 中才有意义。这种表示法表示存在量词这一事实，在 Prolog语义 中有明确规定。另一方面，Oz 方案不需要新的语义。

除了执行所有解的 bagof/3 外，Oz 程序还可以执行惰性 bagof/3，即按需计算每个新解。惰性 bagof/3 由 `Solve` 实现，它会返回一个解的惰性列表。

3 确定性逻辑编程

在 Prolog 或其他逻辑语言中编写逻辑程序，包括定义逻辑语义，然后选择一种能提供令人满意效率的操作语义。这遵循 Kowalski方程“算法 = 逻辑 +控制”。逻辑与控制需要取得平衡。逻辑编程的艺术在于平衡两种相互冲突的要求：逻辑语义应当简单，而操作语义应当高效。做得好时，就能在 Prolog [6] 中形成优雅的编程风格。Oz 通过支持确定性和非确定性控制流，支持这种逻辑编程设计方式。例如，在 Prolog 中，我们可以如下定义一个列表追加谓词：

```
append([],L2, L2). append([X|M1],L2，[X|M3]) :-
append(M1, L2, M3).
```

这个定义遵循 Prolog 的操作语义。编译器可以在某些情况下对其进行优化，使其具有确定性，例如，第一个参数绑定到一个列表时。在 Oz 中，我们可以更精确地定义操作语义。下面展示在 Oz 中定义 append 的三种方式。

3.1 非确定性追加

在第一个定义中，我们以非确定性的方式定义 append：

```
过程 {Append L1 L2 L3}choiceL1=nil L3=L2[]
X M1 M3在L1=X|M1L3=X|M3 {Append M1 L2
M3}结束 结束
```

这与 Prolog 具有相同的操作语义。

3.2 确定性追加（第一个版本）

我们给出 append 的另一种定义，它与之前具有相同的逻辑语义，但具有确定性的操作语义：

```
函数 {Append A B}情况 A的 nil那么 B
[] X|As 那么 X|{AppendAs B}结束 结束
```

在这种情况下，参数A 被绑定到一个列表，因此执行是有方向的。Oz 允许使用函数式语法来定义这类内容。

3.3 确定性追加（第二版）

我们再给出另一种定义，它的逻辑语义相同，但具有第二种确定性操作语义：

```
fun {追加 B C}if B==Cthen nilelse case C of X|Cs t
hen X|{追加 B Cs} end end end
```

这个版本的Append 将最后两个参数作为输入，并返回第一个参数作为输出。例如，{Append [3] [1 2 3]} 返回 [1 2 3]。要正确执行，第二个参数必须是第三个参数的后缀。

4 惰性并发函数式编程

Oz 项目的一项重要见解是，Oz 的逻辑编程核心也能支持函数式编程范式。我们分四个步骤来说明这一点：

- 值式函数式编程。Scheme 或 Haskell 等广泛使用的函数式编程语言通过值进行计算。在 Oz 中，只需不使用未绑定逻辑变量即可实现这一点。为支持高阶编程，我们在 Oz 计算模型中加入函数值。变量可以绑定到函数值，从而实现传统的急切函数式编程。

- 带逻辑变量的函数式编程。我们将逻辑变量引入带值的函数式编程中。绑定操作就是合一。这恰好构成了第 3 节所述的确定性逻辑编程范式。这个范式有一个令人惊讶的优点：更多函数定义可以改写为尾递归形式。

- 确定性数据流。我们将并发引入带逻辑变量的函数式编程中。我们允许任何语句在自己的线程中执行，线程通过变量绑定进行同步。这正是并发逻辑编程的同步模型。不过，与并发逻辑编程相比，我们仍然是纯函数式的。

- 惰性确定性数据流。最后一步扩展加入了按需计算。我们添加了一种新的同步操作，用于等待另一个线程等待某个变量被绑定。借助这个新的基本操作（称为 `WaitNeeded`），我们为确定性数据流加入了惰性求值的全部能力。

我们将更详细地考察这四个步骤。

4.1 值式函数式编程

我们通过函数值扩展计算模型。我们允许将变量绑定到词法作用域闭包。这带来了采用急切求值的纯函数式编程。纯函数式编程是高阶编程的基础，而高阶编程又是数据抽象发展的根基，例如面向对象编程、抽象数据类型、组件、模板、元类等。

除了作为数据抽象的基础之外，这种模型还具有很强的形式性质，这些性质与逻辑编程的强大性质有所不同，但又与之类似。其主要性质称为丘奇-罗瑟性质，通常也称为合流性。该性质指出，表达式归约的最终结果与归约过程中所做的归约选择无关。接下来，我们将展示 Oz 如何利用合流性与逻辑变量相结合所产生的协同作用。

4.2 使用逻辑变量进行函数式编程

现在，我们将逻辑变量引入函数式编程。这与第3节中的确定性逻辑编程完全对应，因此这种形式的

函数式编程实际上就是在进行逻辑编程。这种扩展带来了什么好处？首先，现在可以像在 Prolog 中一样，使用部分实例化的数据结构编写函数式程序。

但我们获得的远不止是使用部分数据结构的能力。允许存在未绑定逻辑变量，使得一些在仅基于值的函数式语言中不可能实现的归约顺序成为可能。例如，第3节中定义的两个确定性追加谓词都是尾递归的；如果不进行复杂的程序变换，这在仅基于值的函数式语言中是不可能实现的。

让我们仔细看看，添加逻辑变量究竟如何使函数变为尾递归。我们从上一节中的第一个确定性追加开始。第3节在语法上将其定义为函数，但在 Oz 中，函数只是过程的语法糖。实际上，确定性追加定义为一个过程：

```
过程 {Append A B C}情况
A为 nil则 C=B[ ] X|As则 Cs
于C=X|Cs {AppendAs B
Cs}结束 结束
```

看一下递归调用 `{Append As B Cs}`。这个调用位于绑定`C=X|Cs`之后，该绑定逐步构建追加操作的输出。`C` 绑定到一个新的 cons 单元，其中包含`X` 以及一个新的未绑定变量`Cs`。由于我们将未绑定变量`Cs`传递给递归调用，因此它可以成为尾调用。这在仅使用值的函数式编程中是不可能实现的。

4.3 确定性数据流

我们通过添加线程并使用逻辑变量进行同步，扩展前面的范式。要理解这一范式，我们首先解释逻辑变量的数据流行为。假设我们声明一个逻辑变量`X`。在线程一中，我们执行`X=10`。在线程二中，我们执行`Y=X+1`。由于调度器的存在，我们不知道哪个线程会先运行。如果线程二先被调度，它在尝试执行加法时会挂起。它会等待`X` 被绑定。线程一绑定`X`后，线程二便会再次变为可运行状态，加法也将完成，并把`Y` 绑定为 11。总而言之，这是一种声明式数据流，包含两项基本操作，即绑定逻辑变量，以及等待变量被绑定。

数据流同步是逻辑变量如此重要的一种用途，因此在讲述 Oz 的历史时，我们将它们称为数据流变量 [13]。Oz 社区并非这一概念的发明者；它最早由并发逻辑程序领域开创，而并发逻辑程序是 Prolog 面向并发系统的一项早期发展 [2, 8]。Oz 对这一概念的贡献，在于将其用于纯函数式并发计算和分布式计算，并将其扩展到惰性求值。

在确定性数据流中，任何指令都可以在自己的线程中执行。这样，我们便能定义由并发代理组成的网络，让它们通过流进行通信；我们将流定义为一种逐步构建且尾部可能未绑定的列表。例如，考虑以下顺序函数式程序：

```
fun{Gen L H} {Delay 1000} % 暂停线程执行 1000 毫秒if L>H then
nil else L|{Gen L+1 H} endend

Xs={Gen 1 10} Ys={Map Xs 函数 {$
X} X*X 结束} {Browse Ys}
```

这会计算一个连续整数列表，将该列表中的每个元素平方，并显示结果。为了在交互式会话中跟踪执行过程，我们降低了生成速度，使其每个元素耗时一秒。我们可以通过让生成和映射分别在线程中执行来实现并发：

```
线程Xs={Gen 1 10} 结束 线程Ys={Map Xs 函数 {$
X} X*X 结束} 结束{Browse Ys}
```

这里使用 **thread** 语句创建新线程。并发版本与顺序版本有什么区别？两种情况下计算结果相同，即 `[1 4 9 16 ... 81 100]`。那么有什么不同呢？在顺序版本中，`Gen` 会先计算出整个列表，然后`Map`才开始执行。最终结果会在计算完成后一次性显示，这发生在十秒之后。在并发版本中，`Gen` 和`Map` 会并发执行。每当`Gen` 向其列表添加一个元素时，`Map`就会立即计算该元素的平方，然后才添加下一个元素。结果会随着元素的生成逐步显示，每秒显示一个元素。并发在不改变函数式语义的情况下，将批处理计算转换成了增量计算。

`Gen` 和`Map` 的并发执行可以视为函数式代理，我们将代理定义为一种并发计算，它读取零个或多个输入流，并写入零个或多个输出流。由于逻辑变量，`Gen` 和`Map` 都是尾递归的。这意味着代理以恒定的栈大小执行。因此，将这些计算称为“代理”是合理的。这种并发执行不仅内存高效，而且是纯函数式的。这清楚地展现了逻辑变量与并发之间的协同作用。

4.4 惰性确定性数据流

最后一项扩展为确定性数据流加入了执行惰性求值的能力。在上一节中，我们解释了逻辑变量如何用于数据流同步。现在，我们将这一数据流范式扩展为执行惰性

求值。为此，我们添加一个新的操作 `WaitNeeded`，它对逻辑变量执行按需同步。操作 `{WaitNeeded X}` 会在 `X` 未绑定且没有其他线程等待 `X` 被绑定时，暂停当前线程。否则，如果另一个线程正在等待 `X`，或者 `X` 已绑定，则该操作成功。我们称 `WaitNeeded` 会“等待直到 `X` 被需要”。向语言中添加这一个操作，就能完整定义惰性求值。Oz 引入了语法糖，让程序员可以轻松使用这一功能。例如，我们定义函数 `{Ints N}`，它返回一个从 `N` 开始、由连续整数构成的惰性列表：

```
函数 惰性 {Ints N}
N|{Ints N+1}结束
```

“`lazy`”注解表示该函数采用惰性求值。这是以下过程的语法糖：

```
proc{Ints N R}线程
{WaitNeeded R} R=N|
{Ints N+1}endend
```

调用 `S={Ints 1}` 时，会在 `WaitNeeded`调用处挂起，直到另一个线程需要 `S` 的第一个元素来运行。发生这种情况时，`WaitNeeded`调用成功，并且 `R` 会绑定到一个包含一个新元素的列表。`Ints` 的递归调用会继续以这种方式处理后续元素。我们注意到，编译器可以通过复用线程来优化上述定义，从而避免每次递归调用都创建一个新线程。惰性求值与并发的结合至少从 1977 [5], 起就已为人所知，但据我们所知，在数据流范式中定义惰性求值及其与逻辑编程的联系，都是 Oz 首创的。

总结 本节总结了本节介绍的各种范式。首先，我们从一种通过值进行计算的一阶范式开始，并分四步对其进行扩展。第一步，加入函数值，即词法作用域闭包，从而得到采用急切求值的纯函数式编程。第二步，加入逻辑变量，从而得到第3节所述的确定性逻辑编程。另一个好处是，用于计算列表等递归数据结构的函数可以成为尾递归函数。第三步，加入线程，并利用逻辑变量的数据流行为进行同步，从而实现确定性数据流。该范式支持由函数式代理构成的并发网络；这些代理通过共享的列表进行通信，这些列表用作通信通道，我们称之为流。请注意，由于列表函数是尾递归的，这些代理只使用恒定的栈空间。第四步，也是最后一步，我们使用`WaitNeeded`加入按需同步。由此得到的范式同时支持惰性求值和并发，我们称之为惰性确定性数据流。这四种范式都保留了纯函数式编程的强大

合流性，同时也是对支持确定性逻辑编程的语言的保守扩展。由于 Prolog 与 Oz 的一个子集完全相同，我们建议，这些扩展中的某种形式可以成为 Prolog 未来演进的一部分。

5 纯函数式分布式计算

Oz 研究社群很早就意识到，Oz 语言的设计是构建分布式编程系统的良好起点。由于该设计清晰地区分了不可变、数据流和可变语言实体，因此可以采用深嵌入方法，为每种语言实体分别实现相应的分布式算法。Mozart 系统在 1999年首次发布时便已包含分布支持。这一设计包含许多重要创新，包括:

- 深度嵌入。每个语言概念都通过一种分布式算法实现。这意味着，为一种分布结构编写的应用程序可以轻松移植到另一种分布结构，而无需更改源代码。两者唯一的区别在于故障行为和时序。故障行为可以通过模块化方式处理，其灵感来自 Erlang [3, 1] 的相关技术。

- 分布式有理树合一。第 4 节中的并发示例在分布到不同计算节点上运行时，都可以保持源代码不变。并发流水线会转变为异步分布式流水线，同时保持纯函数式编程的语义。该系统的关键部分是分布式有理树合一算法。首次对分布式合一进行形式化规范并给出证明，正是在这项工作中完成的，并发表于 [4]。

- 全局引用、分布式词法作用域、分布式垃圾回收和迁移状态。这些是这种设计的其他一些特性。

在本节余下部分，我们将重点讨论我们认为在 Oz 分布式计算研究中取得的最重要发现。这一发现与 Oz 的逻辑编程起源密切相关，也可能为 Prolog 带来机遇。

异步消息传递可以是纯粹的。确定性数据流范式及其惰性扩展可以在分布式环境中实现。数据流变量可以在一个计算节点上读取，并在另一个节点上绑定。这会执行分布式同步，其一般形式通过分布式合一实现。这种分布式同步的显著特性是，它具备纯函数式编程的所有强大属性。具体而言，这意味着计算节点之间的异步消息传递可以完全保持纯粹。这一特性在很大程度上仍不为人所知

在分布式计算领域。在我们作为该领域成员的经验中，我们观察到，许多人往往坚持认为异步消息传递本质上是不纯的，但事实并非如此。

过度使用非确定性 如今大多数大型分布式系统都没有利用这一特性。事实上，情况恰恰相反：如今的系统过度使用非确定性。这是构建和调试此类系统仍然困难的主要原因之一。像 Google、Facebook、Twitter 等每天使用的大型分布式系统，必须由专家持续照看。这并非此类系统不可避免的固有属性。在我们看来，这主要是因为这些系统在设计时过度使用了非确定性。此类系统中的每个库，只要拥有自己的应用程序接口（大多数库都是如此），就会成为一个非确定性来源。

纯函数式分布式计算 其中一位作者在 CodeBEAM 2019 上所作的主题演讲《为什么时间是分布式系统中的祸害，以及如何应对》 [11]，解释了过度使用非确定性的问题及一种可能的解决方案。我们提出的解决方案是，将确定性数据流编程作为分布式系统的默认范式，只在需要之处引入非确定性，除此之外不再引入。采用这一解决方案后，大多数分布式系统都将以函数式为主，并有限度地使用非确定性。这会大幅简化系统的开发、维护和演进。我们发现，这一解决方案与逻辑编程密切相关，因为使其得以奏效的基本概念是数据流变量，它只是在并发环境中使用的逻辑变量。

逻辑编程的作用 我们认为，恰当地扩展 Prolog 或其他逻辑语言，或许能够解决这一问题，并在分布式计算中发挥重要作用。如果逻辑编程社区不进行此类扩展，那么分布式计算社区在解决这些问题时，就会重新发明逻辑编程中的概念。这种重新发明已经随着无冲突复制数据类型（CRDTs）的开发而开始。这些数据类型具有单调性，在许多情况下可以用纯函数式操作取代非确定性的共识算法 [9]。

6 结论

自1972年诞生以来，Prolog一直享有相对广泛的知名度，直到今天亦是如此；相比之下，Oz尽管在1999年发布时包含许多创新，却自2009年起逐渐衰落，如今仅用于教育。Oz的衰落源于社会学方面的原因，与其创新毫无关系；事实上，我们观察到，尽管Oz走向衰落，这些创新却都正逐渐被现代编程系统所采用。作为Prolog和Oz两个社区的成员，

我们认为这是帮助Prolog不断发展的机会，以维持其受欢迎程度。

本文介绍了 Oz 的三项创新：确定性逻辑编程、惰性并发函数式编程，以及纯函数式分布式计算。我们认为，将这些创新加入 Prolog 并不困难，因为 Oz 和 Prolog 的核心逻辑语言十分相似。大部分纯 Prolog 程序都可以直接进行语法翻译，转换为 Oz。这意味着，在逻辑编程语境下构思并理解这些创新的艰苦工作已经完成。我们希望，本文提出的这些观点既能助力 Prolog 的未来发展，也能让 Oz 重新获得作为创新源泉的认可。

参考文献

[1] Francesco Cesarini 和 Steve Vinoski。《使用 Erlang/OTP 进行可扩展性设计：实现健壮且容错的系统》。O'Reilly Media Inc.，2016年。[2] Keith L. Clark。《PARLOG：语言及其应用》。载于 A. J. Nijman、J. W. de Bakker 和 P. C. Treleaven 编，《欧洲并行体系结构与语言会议（PARLE）论文集》。第二卷：并行语言，LNCS 第259卷，第30–53页，荷兰埃因霍温，1987年6月。施普林格。[3] Rapha¨el Collet。《分布式编程语言中网络透明性的局限》。博士论文，鲁汶天主教大学计算机科学与工程系，2007年12月。[4] Seif Haridi、Peter Van Roy、Per Brand、Michael Mehl、Ralf Scheidhauer 和 Gert Smolka。《用于分布式计算的高效逻辑变量》。载于 ACM《编程语言与系统汇刊》，21(3):569–626，1999年5月。[5] Gilles Kahn 和 David B. MacQueen。《协程与并行进程网络》。载于 IFIP大会，第993–998页，1977年。[6] Richard A. O'Keefe。《Prolog 的艺术》。麻省理工学院出版社，1990年。[7] Vijay Saraswat 和 Martin Rinard。《并发约束编程》。载于 POPL 90，第232–245页，1990年1月。[8] Ehud Shapiro。《并发 Prolog 的一个子集及其解释器》。技术报告 TR-003。新一代计算机技术研究所（ICOT），马萨诸塞州剑桥，1983年1月。[9]Marc Shapiro、Nuno Preguiça、Carlos Baquero 和 Marek Zawirski。《无冲突复制数据类型》。载于《自稳定系统研讨会》，2011年，第386-400页。

[10] Leon Sterling 和 Ehud Shapiro。《Prolog 的艺术：高级编程技术》。逻辑编程系列。麻省理工学院出版社，马萨诸塞州剑桥，1986年。[11] Peter Van Roy。《为什么时间是分布式系统中的祸害，以及如何应对》。主题演讲，CodeBEAM 2019，瑞典斯德哥尔摩，2019年5月16日至17日。视频链接：https://www.youtube.com/watch?v=NBJ5SiwCNmU。[12] Peter Van Roy 和 Seif Haridi。《计算机编程的概念、技术与模型》。麻省理工学院出版社，马萨诸塞州剑桥，2004年。课程网站：见 ctm.info.ucl.ac.be/en。[13] Peter Van Roy、Seif Haridi、Christian Schulte 和 Gert Smolka。《Oz 多范式语言的历史》。载于《第四届ACM SIGPLAN编程语言历史会议（HOPL IV）论文集》，《ACM编程语言论文集》（PACMPL），4(83)，2020年6月。