# 对 R语言 向量、数组和矩阵进行依赖类型标注

John Wrenn、Anjali Pal、Alexa VanHattum、Shriram Krishnamurthi

布朗大学
shriram@brown.edu

arXiv:2304.04265v1 [cs.PL] 9 Apr 2023

## 摘要

R语言广泛用于大规模数据分析。它尤其提供了丰富的内置支持，用于处理向量、数组和矩阵。这些操作在构成R语言存在意义的应用中占据重要地位，因此值得理解其行为。此外，表面上是为了方便程序员，它们在 R语言中的行为显著扩展了数学中相应的运算，从而为规范描述和静态验证带来了一些挑战。

我们报告了对静态类型化 R 语言这一方面的进展。在本例中，类型系统有趣之处在于，它可以提醒程序员不要违反边界，因此类型必然需要是依赖类型。我们解释了 R 如何扩展标准的数学行为。随后，我们展示了如何在 LiquidHaskell 中指定R 的行为；LiquidHaskell 是 Haskell的依赖类型扩展。在一般情况下，实际验证库代码和客户端代码目前仍超出了 LiquidHaskell 的能力范围；因此，这项工作既为 R的类型化带来了挑战和机遇，也为依赖类型编程语言的发展带来了挑战和机遇。

## 1. 为 R语言 改造类型

R编程语言 [1]广泛用于 [2] 科学计算应用。与具有深厚语义传统的语言（例如典型的函数式语言）相比，R也是一种语义 [8] 颇为独特的语言。R的不寻常行为已被证明是性能方面的一项挑战，也是编译器和运行时系统研究的主题 [6]。

本文转而聚焦于 R语言 的静态验证。由于 R语言 是一种动态语言，我们遵循一种称为改造 [7] 的设计原则：尽可能将动态错误转化为静态错误。这也是数十年来动态语言研究所遵循的原则，只是针对不同语言进行了不同程度的定制。具体而言，本文关注的是边界检查。

我们的边界检查工作嵌入在更广泛的背景中。我们的类型系统支持多种 R语言 数据类型，因此也可以捕获其他标准错误，例如在函数应用位置错误地使用非函数值。由于以静态方式修复这类错误是众所周知的，而且我们并未对这些错误在 R语言 中有何不同提供任何有趣的见解，因此本文不再进一步讨论这些错误。不过，需要注意的是，本文中的静态信息嵌入在一个传统的类型系统中，该类型系统也能捕获这类错误。

本文其余部分将介绍以下贡献：

1. 为帮助不熟悉 R语言 的读者入门，我们概述了 R语言 的一些特殊行为。这让我们找出希望静态记录的三类信息：形状（如预期）、模式和容器。2. 我们说明了模式和容器约束，并展示如何将它们嵌入现有的类型系统（Haskell 的类型系统）中。3. 我们说明了形状约束，并展示如何将它们嵌入现有的类型系统（LiquidHaskell 的类型系统）中。二者共同为 R语言 的这些部分提供了新的静态语义。4. 我们简要介绍了一套可运行的工具链，它能够将部分 R语言 程序转译为使用上述类型的程序。

5. 随后，我们讨论这些类型规范给 LiquidHaskell 带来的困难。在大多数情况下，无论是库还是客户端都难以进行类型标注，因此这对依赖类型技术构成了一项研究挑战。

6. 我们还讨论了如何进一步强化这些规范。

本文介绍了截至 2016年底这项工作的进展。此后 Liquid Haskell 的改进无疑将使我们能够以更加宏大的目标开展这项工作。

[移除“preprint”选项后，版权声明将显示在此处。]

# 2. R语言向量、数组和矩阵入门

R语言向量操作令人惊讶的灵活性留出了很大的解释空间；这些解释看似严谨，却会被简单的反例所推翻。请看以下表达式：

```
foo <- function(a,b) c(a, b)
```

这里，名称 `foo` 绑定到一个接收两个参数的函数值。函数 `c` 是R语言的内置函数，用于将值组合成一个向量。例如，表达式 `c(1, 2, 3)` 会生成一个包含三个元素的向量：1、2和3。

鉴于在线教程中经常出现类似示例 [3],，R语言初学者可能会认为 `foo` 是一个接收两个参数并生成包含这些元素的向量的函数。在 R REPL 中尝试几个表达式，可能会进一步强化这种理解。调用 `foo`并传入参数1 和 2 会返回一个包含这些值的向量：

```
> foo(3, 4)
[1] 3 4
```

类似地，如果我们向 `foo` 传入两个向量，就可以对其结果连续进行两次下标索引：

```
> foo(c(3,4),c(5,6))[1][1]
[1] 3
```

然而，这些直觉无法解释为什么应用 `foo` 实际上可能返回任意长度的向量，也无法解释为什么可以对应用 `foo` 的结果进行无限次索引（例如，`foo(c(3,4),c(5,6))[1][1][1]...`）。因此，我们将在本节余下部分更详细地解释 R语言的行为。这是对 R语言子集行为的部分描述，并为后续章节讨论的行为形式化编码奠定基础。

## 2.1 模式

R语言中的所有值都带有一个关联的存储模式。这些模式类似于传统语言中的类型，但由于我们将采用更丰富的类型概念，因此会沿用 R语言的术语，将它们称为模式。内置函数 `mode` 可用于检查值的存储模式：

```
> mode(1)
[1] "numeric"
> mode(TRUE)
[1] "logical"
> mode("foo")
[1] "character"
> mode(function(a)a)
[1] "function"
```

在本文中，我们考虑 R语言 的一个子集，其中包括 `numeric`、 `logical`、 `character` 和 `function` 模式，分别表示实数、布尔值、字符串和函数。此外， `numeric`、 `logical` 和 `character` 模式包含常量 `NA`，用于表示缺失数据。

## 2.2 容器

向量程序员可能认为是原始值的内容，实际上往往是一个只包含一个元素的向量。一维、同质的数据集合是 R语言表示数据的通用方式。R语言源代码中的数值型、字符型和逻辑型字面量实例，可以理解为通过语法糖展开，变成包含这些值的长度为一的向量；例如，数值型字面量 `1` 是一个包含分量1的向量。R语言的值在各项操作中始终封装在向量中，因此可能会出现一些意想不到的情况，例如对字面量进行无穷次下标操作：

```
> 5
[1] 5
> 5[1]
[1] 5
> 5[1][1][1][1][1]...
[1] 5
```

考虑到向量装箱，不应将组合函数 `c`视为向量构造函数，而应视为向量连接函数，因为它的参数本身就是向量。例如，前面提到的表达式 `foo(c(1,2), c(3,4))` 求值后得到的是一个长度为四的向量，而不是长度为二的向量。

R语言的常量 `NULL` （不要与 `NA`混淆）是空向量：

```
> c() == NULL
[1] TRUE
> length(NULL)
[1] 0
```

在本文中，我们作出一个简化假设：向量具有固定大小。

数组 `array` 类型提供了在 R语言 中表示n 维数据的功能。数组对象充当包装器，封装了一个包含数组内容的向量，以及一个用于描述数组维度的 n 个数值组成的向量。因此，数组使用两个向量构造：

```
> array(c(1,2,3,4), c(2,2))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

如果程序员没有提供足够的元素来填充指定的形状，这些元素会被循环使用以弥补差额。

```
> array(c(1,2), c(3,3))
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    2    1    2
[3,]    1    2    1
```

矩阵 `matrix` 类型用于表示二维数据，可使用 `matrix`函数构造：

```
> matrix(c(1,2,3,4),c(1,4))
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

在行为上，矩阵可以被视为仅仅是赋予二维数组实例的一个名称：

```
> class(matrix(c(1,2,3,4),c(1,2)))
[1] "matrix"
> class(array(c(1,2,3,4),c(1,2)))
[1] "matrix"
```

## 2.3 二元运算

R语言的二元算术运算（+、 -、 *、 /等）和二元逻辑运算（&、 |、 >、 ==等）会对其参数的元素逐对执行相应的运算：

```
> 1 + 1
[1] 2
> c(1, 2, 3) + c(1, 2, 3)
[1] c(2, 4, 6)
```

混合容器之间二元运算的左操作数和右操作数可能属于不同的容器类型。如果两个操作数都是向量，则运算结果为向量；如果任一操作数是数组，则结果为数组。

在混合模式之间 在二元逻辑运算符中，数值参数会被强制转换为逻辑值（零会被视为假），结果始终是逻辑值的集合。

```
> TRUE & 0
[1] FALSE
> TRUE & -2
[1] TRUE
```

在二元算术运算符中，逻辑值会被强制转换为数值型值（TRUE 会被强制转换为1，而 FALSE 会被强制转换为0）。

```
> 1 + FALSE
[1] 2
> 1 + TRUE
[1] 0
```

字符值不能被强制转换为逻辑或算术模式。

```
> 1 + "R"
Error in 1 + "R" : non-numeric
argument to binary operator
```

不同大小之间当容器之间二元运算的操作数长度不同时，较短操作数的元素会被循环复用，以弥补长度差异。表达式：

```
c(1, 2, 3, 4) * c(0, 1)
```

可以理解为在执行两两运算之前，先进行一个中间步骤：

```
> c(1, 2, 3, 4) * c(0, 1, 0, 1)
[1] 0 2 0 4
```

如果较长的操作数不是较短操作数的整数倍，R语言会发出警告：

```
> c(1,2) + c(1,2,3)
[1] 2 4 4
Warning message:
In c(1, 2) + c(1, 2, 3) :
  longer object length is not a
  multiple of shorter object length
```

## 2.4 一维索引

向量和数组都支持一维索引操作。以一维方式索引多维数组时，会忽略数组的维度规则，并对数组内部的一维内容向量进行索引。

```
> array(c(1,2,3,4), c(2,2))[3]
[1] 3
```

使用正数当使用一个由正数构成的容器对容器进行索引时，R语言会返回一个向量，其中包含位于与每个正数相对应的索引处的值。这种索引方式例如可用于从输入向量生成一个新向量，使该输入向量中的值以不同顺序排列：

```
> c("a", "b", "c")[c(3, 2, 1)]
[1] "c" "b" "a"
```

使用负数时 当使用由负数构成的容器进行索引时，R语言返回一个排除这些索引处所有值的向量。

```
> c("a", "b", "c")[c(-1, -3)]
[1] "b"
```

使用逻辑值 当使用逻辑值容器对容器进行索引时，R语言会将逻辑值容器作为掩码应用于向量，返回一个容器，其中包含每个 TRUE 值下方的元素。如果逻辑向量短于被索引的向量，其值会循环重复，以填补长度差异。这种用法的一个常见示例是屏蔽向量中的每隔一个元素：

```
> c("a", "b", "c")[c(TRUE, FALSE)]
[1] "a" "c"
```

## 2.5 下标赋值

可以使用下标赋值运算符修改容器，其行为基本遵循带赋值的单维索引。当替换向量短于要替换的元素数量时，替换向量中的元素会循环重复，如第三次 REPL 交互所示：

```
> a <- c(1, 2, 3, 4)
[1] 1 2 3
> a[c(1,2)] = NA; a
[1] NA NA  3 4
> a[c(1,2,3,4)] = c(1, 2); a
[1] 1 2 1 2
```

与成对二元运算一样，当右操作数不是左操作数的倍数时，下标赋值操作会发出警告：

```
> a[c(1,2)] = c(NA, NA, NA); a
Warning message:
In a[c(1, 2)] = c(NA, NA, NA) :
  number of items to replace is not
  a multiple of replacement length
[1] NA NA  3
```

当右操作数是 NULL 时，R语言会报错：

```
> a[1] = NULL
Error in a[1] = NULL :
  replacement has length zero
```

## 3. 初步的 R语言类型检查工作流

类型的本质 这些示例涵盖了 R 语言中一个相对较小但内容丰富的子集，形式化编码和自动验证工具将对此大有帮助。此外，这些示例也指向了 R 的类型这一概念。容器之间二元运算的限制似乎可以表示为操作数各自的在线视频精品

- 容器类型，
- 存储模式，和
- 大小。

这些属性对二元运算结果类型的影响各不相同。二元运算操作数的容器类型要求该二元运算必须重载，因为根据操作数的容器类型，运算结果可能是向量，也可能是数组。二元运算是算术运算还是逻辑运算，决定了其结果的存储模式；而根据操作数的模式，可能需要先强制转换一个或两个操作数的模式，才能执行该运算。Haskell 的类型系统足够富有表现力，能够编码类型与行为之间的这种相互作用。

然而，二元运算操作数的大小会对操作数的大小施加约束，并决定运算结果的大小。要编码这些约束，需要依赖类型系统所具备的表达能力。我们使用 LiquidHaskell 的精化语言作为规范语言，用于说明大小约束。

目标语言选择 选择 LiquidHaskell 是出于实用考虑。原则上，我们只需要一种能够表达这些约束的依赖类型语言；事实上，许多其他方案也可能同样适用，甚至效果更好。不过，使用 LiquidHaskell 有两个好处：

- 我们可以利用 Haskell 类型系统的表现力。它本身已经非常强大，并且能够编码 R语言 的多项特性；例如，它的类型类可以自然地表达 R语言 的重载机制。
- LiquidHaskell 是对 Haskell 的扩展，作为一种语言，它日益成熟，并具备完整的检查器自动化能力。对于从另一种语言开始的工作流程而言，自动化至关重要。如果改用交互式证明助手，那么 R语言 程序员就得为经过大幅改写的程序编写证明步骤（例如，以状态单子方式编码），而所用语言与 R语言 毫无相似之处。

> 特别是，LiquidHaskell 的自动化构建于 SMT [4]（可满足性模理论）之上。这些理论为检查器提供了一个可扩展的框架。如果我们发现 R 中某些模式无法自然地进行类型检查，原则上可以向求解器添加理论，以解决这些模式。从这个意义上说，我们可以将 LiquidHaskell 的精化视为一种便捷的接口，用于调用 SMT 求解器解决 R 的数组约束，而不必自行构建这种转换。

当然，这些论证并非规范性的：它们并不排除使用其他依赖类型化目标语言的可能性。此外，这一翻译显然引发了一些非平凡的用户界面问题，而如何让错误信息能够被 R语言程序员解读，仍然是一个尚未解决的问题。尽管如此，Haskell与 LiquidHaskell 的结合为解决方案的原型设计提供了良好的基础，之后或许可以将该方案转译为 R语言本身的原生解决方案。

该流程这些规范使我们得以开始构建LiquidR，这是一个用于对部分 R语言程序进行类型检查的初步流程。我们的系统会解析 R语言程序，将其化简为核心语言，然后再将该化简结果编译为Haskell。编译后的结果使用了一个替代性的前导库，该库实现了许多属于我们所支持的 R语言子集的数据类型和操作。根据后续章节中描述的规范，原则上我们可以执行两类检查：

1. 最重要的是，我们可以通过编译 R 程序，并对结果应用 Haskell 和 LiquidHaskell类型检查器，来检查该程序是否违反这些类型约束。这假设我们的规范足够忠实地反映 R语言的语义（我们会通过探索大量示例来尽力确保这一点）。
2. 此外，我们可以检查在 Haskell 中重新实现的 R 的向量、数组和矩阵运算是否忠实地实现了这些规范。当然，这只会让希望在 Haskell 中使用 R 语义的 Haskell 程序员感兴趣。

请注意，前者检查客户端（使用 R语言内置操作的 R语言程序），而后者检查服务器（这些操作的 Haskell 实现）。这两者各自都有独立的价值。当然，使用 R语言操作的 Haskell 实现的 LiquidHaskell 用户，会将这些规范用于这两个目的。

由于 LiquidHaskell 的局限性，我们在这两个方面都只取得了初步进展，详见第 5.1 节。

## 4. 在（Liquid）Haskell 中编码 R语言

每种具体模式的类型都用 Haskell 的`Maybe` 类型编码，该类型有两个变体： `Just` （用于表示存在一个值）和 `Nothing` （用于表示不存在值）。我们用 `Nothing` 来编码 R语言的 `NA`:

```
type Logical   = Maybe Bool
type Numeric   = Maybe Double
type Character = Maybe Double
```

为了将这些类型定义为模式，我们创建并实例化一个类型类 `Mode`:

```
class Mode a
instance Mode Logical
instance Mode Numeric
```

给定模式的单个值之间的每种运算，都定义为描述该运算的类型类的一个实例:

```
class (Mode a, Mode b, Mode c)
    => Addition a b c | a b -> c
  where mode_add :: a -> b -> c
```

例如，将两个数值型相加，结果要么是一个整数，要么在任一参数为 `Nothing`时，结果是一个 `NA`:

```
instance Addition Numeric
                  Numeric
                  Numeric where
  mode_add (Just l) (Just r) = Just (l + r)
  mode_add _ _ = Nothing
```

### 4.1 容器表示

向量 向量被编码为 Haskell 列表:

```
type Vector a = [a]
```

数组我们将数组表示为一对列表，第一个列表存储数组的维度，第二个列表存储数组的成员:

```
type Array  a = ([Int],[a])
```

然而，这种 Haskell 类型无法排除存储概念上有缺陷的数组；例如，元素数量可能少于或多于维度所暗示的容量，数组也可能以负维度实例化。我们使用 LiquidHaskell 细化来确保元素数量等于各维度的乘积:

```
type Array a = ([Nat],[a])<{
    \shape elems ->
      (product shape) = (len elems)
  }>
```

矩阵 矩阵只是二维数组的另一种称呼，在 Haskell 类型层面上无法与数组区分，因为 Haskell 的表达能力不足以将 `Array` 元组的第一个分量约束为包含两个元素的列表。因此，作为一种 Haskell 类型， `Matrix`是 `Array` 的别名:

```
type Matrix a = Array a
```

我们通过扩展针对 `Array` 的 LiquidHaskell细化来编码额外的维度限制:

```
type Matrix a = ([Nat],[a])<{
    \shape elems ->
      (product shape) = (len elems) &&
      (len shape) = 2
  }>
```

抽象表示由于向量、数组和矩阵在许多上下文中可以互换使用，我们使用 Haskell 类型类 `Container` 来刻画容器特性，并为 `Vector` 和 `Array` 都提供其实例:

```
class Container m where
  length :: m -> Int

instance Container (Vector a) where
  length = ...

instance Container (Array a) where
  length = ...
```

### 4.2 容器构造

向量尽管我们使用 Haskell 的列表来编码 R向量，但仅将 R `c` 函数的实例编译为 Haskell 列表字面量是不够的，因为 `c` 函数会连接其参数。 `c`的输出长度是其输入长度之和。我们通过创建 LiquidHaskell 测度 `llen`来编码这一长度概念，该测度接收一个列表的列表，并生成它们长度之和:

```
measure llen :: [[a]] -> Int
  llen ([])   = 0
  llen (x:xs) = (len x) + (llen xs)
```

我们将R语言的 `c` 函数实例编译为 Haskell 函数 `combine`。我们使用 `llen` 测度来细化 `combine` 的结果长度:

```
combine ::
 ls:([[_]]) ->
 {l:([_]) | (len l) = (llen ls)}
```

数组 R语言中的数组使用 `array` 构造函数（而不是字面语法）构造。它接收一个维度向量和一个元素向量，并生成 `array` 类的实例。同样，直接将 `array` 构造函数的用法编译为 Haskell 中的数组表示并不合适，因为使用我们的 `Array`数据类型会导致构造函数受到过多约束。请注意，如果数组构造函数提供的元素足以填满指定的形状，这些元素会被循环复用以弥补差额:

```
> array(c(1,2), c(3,3))
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    2    1    2
[3,]    1    2    1
```

因此，我们将 `array` 构造器的出现转换为对一个名为 `array` 的函数的调用，该函数只约束数组必须非空（通过以下 Liquid-Haskell 细化）：

```
array :: (Mode t) =>
    shape:(NonEmptyContainer NonNegNumeric) ->
    elems:(NonEmptyContainer t) ->
    reslt:(Array t)
```

该约束使得此 R语言运行时错误能够被静态报告：

```
> array(c(),c())Error in array(c(), c()) :
'data' must be of a vector type, was 'NULL'
```

矩阵 向 R语言 `matrix` 构造器指定超过两个维度并不是错误；但是，生成的数组将被截断为给定的前两个维度。如果我们认为此类截断很可能是程序员错误的结果，就可以通过为 Haskell `matrix` 构造器提供一个拒绝多余维度的 LiquidHaskell 细化，在静态检查时报告此问题：

```
matrix :: (Mode t) =>
    shape:(ContainerNElems NonNegNumeric 2) ->
    elems:(NonEmptyContainer t) ->
    reslt:(Matrix t)
```

### 4.3 二元运算

回顾第2.3 节，容器上的运算会对其参数施加三类约束（模式、容器类型和大小）。为了编码对操作数的模式和容器类型的约束，我们为每个二元运算构造一个 Haskell 类型类。在本节中，我们将考虑加法的一种编码：

```
class (Container a,Container b,
Container c)
=> Addition a b c | a b -> c
where add :: a -> b -> c
```

给定操作的容器和模式的所有有效组合，都编码为该操作的类型类实例。例如，我们通过使用这些类型实例化加法类型类，将数值向量与逻辑值数组之间的加法识别为可接受的操作：

```
instance Addition (Vector Mode.Numeric)
                  (Array  Mode.Logical)
                  (Array  Mode.Numeric)
  where add = ...
```

请注意，此实例的参数化方式表示，向量与数组相加时，预期结果为数组。

尽管二元运算不会对操作数施加大小约束，但 LiquidHaskell 能否为使用那些确实施加大小约束的运算的客户端提供保证，取决于它对运算所产生数据的大小进行推理的能力有多强。因此，我们引入一种 LiquidHaskell 细化，用于描述不约束操作数、但会细化结果长度的二元运算。下面的 LiquidHaskell 细化中，`Arithmetic`是一个类型，用于描述这样一个函数：它接收两个非空容器，并生成一个长度等于较长操作数长度的容器。如果任一操作数为空，结果也为空：

```
type Arithmetic t u v =
  (Container t, Container u, Container v) =>
    a:t ->
    b:u ->
   {c:v | if (len a) = 0 || (len b) = 0
          then (len c) = 0
          else if (len a) >= (len b)
            then (len c) = (len a)
            else (len c) = (len b)}
```

二元运算的操作数大小不会产生任何硬错误，但请记住，如果较大操作数的长度不是最短操作数长度的倍数，就会产生警告。我们可以通过使用操作数长度的模数来测试倍数关系，从而强化细化，使其在静态检查时将此警告报告为类型错误：

```
type Arithmetic t u v =
  (Container t, Container u, Container v) =>
    a:t ->
    b:u ->
   {c:v | if (len a) = 0 || (len b) = 0
          then (len c) = 0
          else if (len a) >= (len b)
               && (len a) mod (len b) = 0
            then (len c) = (len a)
          else if (len a) < (len b)
               && (len b) mod (len a) = 0
            then (len c) = (len b)
          else false}
```

### 4.4 下标运算

其他容器操作，例如下标操作，也通过不同的 Haskell类型类进行编码，例如：

```
class (Container a,
       Container b)
      => Subscript a b
  where subscript :: a -> b -> a
```

并且可接受的操作被编码为实例：

```
instance (Container a)
    => Subscript a (Vector Mode.Numeric)
  where subscript = ...
instance (Container a)
    => Subscript a (Vector Mode.Logical)
  where subscript = ...
```

由于我们尚未考虑具有命名维度的数组，因此省略了使用`(Vector Mode.Character)`参数化的实例。与二元运算一样，下标运算不会报告错误（甚至不会发出警告）。尽管如此，我们仍然细化了它们的

结果类型带有预期的大小，以便 LiquidHaskell 可以在程序的其他位置对这些值进行推理。

使用正数的回顾：回想第 2.4 节，当使用由非负数值组成的容器对向量或数组进行索引时，R语言会返回一个容器，其中包含与这些正数分别对应的索引处的值。因此，结果是一个与下标长度相同的容器：

```
type SubscriptNonNegative t =
  (Container t) =>
    a:t ->
    b:(Container NonNegNumeric) ->
   {c:t | (len c) = (len b)}
```

使用负数请注意，当使用包含负数的容器进行下标操作时，R 会返回一个排除这些索引处所有值的向量。R 将下标向量视为一个集合（因此，排除某个元素多次与排除一次等价），并忽略超出数组边界的排除项。结果的长度与下标的长度无关，而是取决于下标中位于数组边界内的不同元素索引的数量。这可以编码为 Liquid-Haskell 细化类型，但会增加约束求解器的负担。因此，更实用的做法是表达一个较弱的约束：负数下标操作产生的输出长度不能超过被进行下标操作的向量长度。

```
type SubscriptNonPositive t =
  (Container t) =>
    a:t ->
    b:(Container NonPosNumeric) ->
   {c:t | (len c) <= (len b)}
```

最后来看逻辑值。回想一下，使用逻辑值容器进行下标操作时，R语言会将该容器用作向量的掩码，并返回一个容器，其中包含每个 `TRUE` 值所对应的元素。如果下标中的 `TRUE` 或 `NA`值数量大于被操作向量的长度，那么使用逻辑值进行下标操作所得到的结果可能会比被操作的容器更长：

```
> c(1,2,3)[c(TRUE,TRUE,TRUE,TRUE)]
[1]  1  2  3 NA
```

如果下标中的 `TRUE` 或 `NA` 值数量少于被操作容器中的值数量，那么此操作所得结果可能会比被操作的容器更短：

```
> c(1,2,3)[c(FALSE,TRUE,FALSE,FALSE)]
[1]  2
```

如果不说明下标内容的性质，就无法对该操作结果的长度进行有意义的细化。

Haskell集成编码索引带来了集成挑战。如果我们使用Haskell类型类重载索引操作，就无法为 `Subscript` 定义两个实例，用于一个 `Numeric` 下标使得这些实例随后可以分别用 `SubscriptNonNegative` 和 `SubscriptNonPositive`进行细化；Haskell 会将这些类型类实例判定为重叠实例而拒绝。由于只能定义一个在下标位置使用 `Numeric` 的 `Subscript` 实例，并且该实例只能用单个LiquidHaskell类型进行细化，因此我们通过将 `SubscriptNonNegative` 和 `SubscriptNonPositive` 精化合并为单个精化来细化此实例：

```
type SubscriptNumeric t =
  (Container t) =>
    a:t ->
    b:{bv:(Container Numeric) |
        (allNonNegative b)
     or (allNonPositive b)} ->
   {c:t | if (allNonNegative b)
          then (len c) == (len b)
          else if (allNonPositive b)
          then (len c) <= (len b)
          else false}
```

### 4.5 下标赋值

与前面的操作一样，我们使用 Haskell类型类及相应的实例来编码下标赋值操作中可接受的模式与容器组合。

```
class (Container a,Container b,Container c,
Container d)
=> SubscriptAssignment a b c d | a b c -> d
where subscriptAssign :: a -> b -> c -> d
```

虽然下标赋值在很大程度上沿用了下标的行为，但它还引入了一个警告和一个与替换值大小有关的错误。我们将依次讨论每种情况：

对于正数，这种表达能力足以静态报告倍数关系警告和 NULL 赋值错误：

```
type SubscriptAssignmentNonNeg t v =
 (Container t, Container v) =>
   a:t                          ->
   b:(Container NonNegNumeric) ->
  {c:v | (len c) > 0
         if (len a) >= (len b)
           then ((len a) mod (len b)) = 0
           else ((len b) mod (len a)) = 0} ->
  {d:t | if (len b) > (len a)
           then (len d) = (len b)
           else (len d) = (len a)}
```

然而，只有在假设向量具有固定大小的情况下，这种说法才准确。事实上，结果的长度等于 `b` 中最大的数值。

对于负数，不会报告倍数关系警告；要做到这一点，必须知道 `a` 的索引集合与 `b` 中各成员绝对值所构成集合之间的差集。

```
type SubscriptAssignmentNonPos t v =
```

```
 (Container t, Container v) =>
   a:t                          ->
   b:(Container NonPosNumeric) ->
  {c:v | (len c) > 0 }          ->
  {d:t | (len a) == (len b)}
```

对于逻辑值，不报告倍数关系警告；要报告此警告，需要知道 `b` 掩码了 `a` 中的多少个元素。

```
type SubscriptAssignmentLogical t v =
 (Container t, Container v) =>
   a:t                          ->
   b:(Container Logical)        ->
   c:(Container NonPosNumeric) ->
  {d:t | (len a) == (len b)}
```

## 5. 从 Here 到 TheRe

### 5.1 流水线开发

在撰写本文时，我们已能够使用 R语言到 Haskell 的流水线，在这个子集中交叉编译适度规模的 R语言程序。对于本文所述规范的各种变体，我们还成功地使用该流水线，在静态阶段揭示 R语言中的基本运行时错误。如下所示的 R语言程序

```
foo <- function(a,b) c(a,b)
a <- foo(1,c(NA,3))
b <- foo(c(1,NA),3)
a + b
```

它的计算结果是一个包含 `2`、 `NA`、`6` 的向量，将被转译为以下 Haskell 程序:

```
import LiquidRprogram = do
let foo = \\(x,y) -> combine [x,y]
let a = foo ([Just 1.0],
combine [[Nothing],[Just 3.0]])
let b = foo (combine [[Just 1.0],
[Nothing]],[Just 3.0])a `add` b
endOfProgram
```

Haskell 类型检查 使用我们的编译流程，本文提出的 R语言向量和数组类型的 Haskell 编码，足以静态报告基本的模式不匹配和函数元数不匹配。运行我们编译流程对以下程序的编译结果时，Haskell 类型检查器能够针对 R语言的每个动态错误报告相应的 Haskell 类型检查错误:

```
> foo(1)
Error in a(1, 2, 3, 4):
  unused arguments (1, 2, 3)
> foo(1,2,3)
Error in foo(1, 2, 3):
  unused argument (3)
> "bar"(1,2)
Error:
  attempt to apply non-function
> "cat" + 1
Error in "cat" + 1 : non-numeric argument to
    binary operator
```

LiquidHaskell 类型检查 我们正在将 LiquidHaskell 规范整合到这一编译流程中。遗憾的是，尽管我们的 Haskell 类型编码高度依赖类型类，但 LiquidHaskell 对类型类精化的支持是近期才加入的，目前仍处于初步阶段。我们发现了若干个错误，导致无法为类型类上的二元运算添加注解，目前正在向 LiquidHaskell 团队报告这些问题。

我们将这些精化的一种修改版本整合到一个精简的标准库中，该标准库不支持运算符重载（例如，其中唯一的容器是向量，唯一的模式是数值型），并使用 LiquidHaskell 类型检查器静态报告基本的大小相关错误。

### 5.2 扩大范围

我们设想将这项工作的范围拓展到三个层次，其中第一个应当相对直接，第二个需要大量工作但看起来可行，而第三个几乎不可能实现。

1. 最自然的扩展是涵盖向量、数组和矩阵的更多约束。我们已经讨论了其中一些限制。我们尚未探索这些内容的主要原因，是让 LiquidHaskell完成类型检查存在困难，因此我们对更丰富断言的正确性缺乏信心。一旦修复该系统中的错误，并且我们能够检查这些断言，我们认为，更精确地捕捉 R语言的行为并不存在重大障碍。

2. 下一步自然是涵盖更多R语言 的数据类型。有些类型（例如函数）我们目前只进行了部分涵盖；虽然我们能够捕获某些函数错误，但R语言 的函数还有许多特性——例如可选参数、命名参数、缩写形式的命名参数 [9]——我们尚未涵盖。此外，R语言 还有若干其他特性，例如数据框（名称与数据的多维关联），或R语言 的三种不同对象系统。自然，对其中每一项进行建模都需要付出大量努力。

3. 然而，最微妙的问题在于如何对核心语言本身建模。在描述编译器时，我们隐含地假设 R语言具有固定的语言定义，因而可以采用静态编译系统。大多数语言都符合这一假设，但 R语言并非如此。

> 在 R语言中，即使是被认为是关键字和内置结构的内容，如 `if`、函数定义和赋值，实际上也都是函数，而初始库会将它们绑定到特定的定义。由于它们只是被绑定的变量，它们的

绑定可以通过突变进行更改，并且它们会被全局重新定义。
我们将这种行为称为反语法的，因为该语言实际上处处设法阻碍由语法驱动的静态处理。因此，我们的编译器假定语言构造具有默认含义，且未被重新定义。要创建一个能够在语法发生变化时正确建模 R语言真实行为的静态类型系统，似乎是一项相当艰巨的任务。

近期，我们计划改进编译流程以及用 Haskell 编码的 R语言类型，以建模下标赋值操作的变更效应。向量及向量操作在 R语言中无处不在，这是该语言实现惯用代码的诀窍之一；另一诀窍是将名称与值关联起来。我们希望探索如何转译使用具名参数函数的 R语言程序。向量和数组的维度也可以命名；尽管这会大幅增加精化的复杂度，但我们认为 Liquid-Haskell 的表达能力足以应对。解决这些问题或许能为数据框提供有效的编译策略。

## 6. 相关工作

尽管已有大量关于脚本语言的研究，其中一些研究为我们提供了启发，但大多数研究并不直接适用于本文。同样，关于 R语言运行时系统性能改进的文献通常是在相当低的抽象层次上表述的，明显低于本文中的规范。

Morandat 等人 [8] 描述了 R语言一个规模较大但仅部分重叠的子集的形式语义。他们的工作没有深入探讨 R语言向量和数组运算的细节。其形式语义没有对数组、逻辑模式值和字符模式值进行建模，并且只考虑单维的非负数值索引，其中下标是长度为一的向量。因此，我们将本文视为对其工作的补充，为 R 语言的另一个角落提供部分语义。此外，我们还投入了大量精力，将这些语义编码到一个求解器中，并着眼于构建类型检查流程，而他们的工作并未涉及这一点。

Slepak 等人的基于数组的 [10] 类型语言 Remora是一种从零开始设计的数组编程语言。因此，Remora 提供了一种优雅的核心语言，可以作为R语言子集的良好翻译目标。然而，Remora 不支持突变，限制了其适用性，并且在索引方面存在局限。此外，它目前尚未提供强大的求解器支持。因此，尽管它是一个很有前景的平台，我们还是选择不将其用作编译目标。

## 7. 结论

本文代表了对静态类型化 R 的一种初步尝试。基于向量、数组和矩阵等关键 R 构造的行为，我们创建了一组依赖类型规范，用于捕捉其中的大部分行为。我们还创建了一个初步的流水线，用于处理 R 源程序，将其编译为 LiquidHaskell，并自动运行类型检查器。

遗憾的是，目前我们的流程仅适用于非常小型的程序。我们遇到的主要困难在于 LiquidHaskell：由于我们的翻译需要使用该语言中全新的功能，我们碰上了一些未经测试且存在错误的边缘情况。因此，我们尚不清楚修复这些错误后，这些断言能否得到快速检查——不过我们的一些其他断言还无法快速检查——也不清楚这项任务是否会给依赖类型求解器带来新的挑战（可能需要扩展底层的可判定理论）。

无论如何，我们认为这项实验在多个方面都很有用：有助于为 R语言添加类型、检验 LiquidHaskell（向其提出与作者设想的问题明显不同、足以触发错误的问题），也可能有助于检验求解器的能力。尤其是，在后两个方面取得进展，从而为 R语言添加类型，将对许多同样依赖这些系统的其他任务产生积极影响。

我们还认为，这个项目对其他语言也有价值。其他一些科学数据处理语言也具有不同寻常的语义，扩展了传统数学的语义；这些语言同样会受益于我们的规格说明。如果它们的语义更易处理，那也不是问题：这只会让规格说明更简单（例如，我们可以直接删去规格说明中的部分内容），也可能加快求解由此产生的约束。因此，值得将我们的规格说明与 SciPy 等库的规格说明进行比较 [5], ；类似于我们的工具链也可能适合编译这些库。

最后，丰富的类型系统还带来了其他可能性。例如，尽管R语言没有原生的“单位”系统，但单位和量纲分析显然是科学计算中至关重要的检查手段。因此，考虑对我们的规范进行扩展，使单位能够与值的其他精化一起进行编码和计算，将会很有意义。

## 致谢

感谢 Alex St. Laurent 澄清了与 Slepak 等人的语义之间的关系，也感谢 Justin Pombrio 校对并提供反馈。

# 参考文献

[1]R 统计计算项目，2016 年。网址：https://www.r-project.org/.[2] R 在学术研究中的使用超过 SAS，2016 年 6 月。网址：http://r4stats.com/2016/06/08/r-passes-sas-in-scholarly-use-finally/.

[3]R 教程：组合向量，2016 年。URL http://www.r-tutor.com/r-introduction/vector.[4]R. Jhala。Haskell的精化类型。载于ACMSIGPLAN 2014 年编程语言与程序验证相遇研讨会论文集，PLPV'14，第27–27页，美国纽约州纽约，2014 年。ACM。ISBN 978-1-4503-2567-7。doi: 10.1145/2541568.2541569。URL http://doi.acm.org/10.1145/2541568.2541569。[5]E.Jones、T. Oliphant、P.Peterson等。SciPy：Python 的开源科学工具，2001 年至今。URL http://www.scipy.org/。[6]T.Kalibera、P. Maj、F. Morandat 和 J. Vitek。R 的快速抽象语法树解释器。载于第 10 届 ACM SIGPLAN/SIGOPS 虚拟执行环境国际会议论文集，VEE '14，第 89–102页，美国纽约州纽约，2014 年。ACM。ISBN978-1-4503-2764-0。doi: 10.1145/2576195.2576205。URL http://doi.acm.org/10.1145/2576195.2576205。[7] B. S.Lerner、J. G. Politz、A. Guha 和 S. Krishnamurthi。TeJaS：为JavaScript 改造类型系统。载于动态语言研讨会，2013 年。[8]F. Morandat、B. Hill、L. Osvald和 J.Vitek。评估R 语言的设计：用于数据分析的对象和函数。载于第 26 届欧洲面向对象编程会议论文集，ECOOP'12，第 104–131 页，柏林、海德堡，2012 年。Springer-Verlag。ISBN978-3-642- 31056-0。doi:10.1007/978-3-642-31057-76。[9] T.Mortel。奇妙的异类：R 的函数调用语义，1 2006 年。URL http://blog.moertel.com/posts/2006-01-20-wondrous-oddities-rs-function-call-语义.html。

[10]J. Slepak、O. Shivers 和 P. Manolios。《一种支持静态秩多态的数组导向语言》，第 27–46 页。Springer Berlin Heidelberg，柏林，海德堡，2014 年。ISBN 978-3-642-54833-8。doi: 10.1007/978-3-642-54833-8 3。URL http://dx.doi.org/10.1007/978-3-642-54833-8_3。