JFP，32页，2023。©剑桥大学出版社2023
doi:10.1017/xxxxx

arXiv:2305.04480v1 [cs.PL] 8 May 2023

Idris TyRE：一个依赖类型的正则表达式解析器

OHAD KAMMAR KATARZYNA MAREK
苏格兰爱丁堡大学 电子邮件：{
kmarek,ohad.kammar}*@ed.ac.uk*

摘要

正则表达式（regex）不仅广泛用于验证，也广泛用于解析文本数据。通常，正则表达式解析器输出的是松散的结构，例如由匹配项组成的非结构化列表，留给用户验证输出的属性，并将其转换为所需的结构。由于正则表达式本身携带了有关结构的信息，这种设计造成了不必要的重复。

Radanne 提出了类型化正则表达式——TyRE——一种可添加到现有正则表达式引擎之上的类型索引组合子层。我们扩展了 Radanne 的设计，并实现了一个解析器，使所有层都能始终保持类型安全：面向用户的正则表达式、其内部的去糖表示、编译后的有限状态自动机，以及该自动机用于构造解析树的关联指令集。我们使用依赖类型语言 Idris 2 实现了 TyRE。

1 引言

正则表达式是一门历史悠久且研究充分的领域。对于一类使用场景而言，正则表达式已成为验证或解析文本数据的标准方法。然而，在静态类型语言中，正则表达式解析器并不具备手写解析器或基于组合子的解析所能提供的同等类型安全性。尽管模式——即正则表达式——在编译时已知，类型系统却没有利用其中包含的信息来约束结果。例如，考虑编程语言 Scala 2 标准库中的正则表达式解析器。使用正则表达式解析字符串会得到一个由匹配项组成的元组，用户可以通过模式匹配调整其形状。不过，确保元组基数正确仍由用户负责，尽管编译时已有所有必要信息可用于检查这一基数。

这里我们介绍一种正则表达式解析器，它会生成模式依赖类型的结果。这种设计消除了元组基数错误等问题。与解析器组合子一样，结果的形状反映了解析器的结构。这种类型机制既能确保解析器本身的正确性，也能确保模式的正确性。此外，用户可以像使用组合子那样编写结果数据转换，以获得所需的数据结构，而无需重复模式中包含的信息。但与解析器组合子设计不同，该解析器的时间和空间复杂度均保证相对于所解析字符串的长度呈线性。

本工作基于 Radanne（2019）设计的带类型正则表达式解析器。Radanne 利用现有的正则表达式引擎，并在其上添加一层带类型的封装。该层增加了

安全性，并支持便捷的数据转换。他使用 OCaml 编程语言实现了这一设计（Leroy 等人，1996），并使用 Vouillon 的正则表达式引擎[1]。

Idris 2（Brady，2021）是一种新型严格依赖类型编程语言。Idris 2 没有标准的正则表达式解析器库，这为重新审视 Radanne 的设计提供了机会。新设计在所有层级中都保持类型安全，使类型约束成为对解析器本身的额外安全检查。

我们的贡献如下：

- Moore 机的类型化指令语言；
- 将编译方案转换为这种机器语言，作为 Thompson 正则表达式到非确定性有限状态自动机构造的配套方案；
- 一个使用类型级解析器的面向用户的正则表达式字面量层；以及
- 一个用于实际场景的流编辑库——*sedris* ——使用新的 TyRE 库执行搜索和替换操作。

Th这两个库（正则表达式[2] 和流编辑[3] ）的源代码均已公开发布。

本文结构安排如下。第2 节简要介绍该库的核心功能。第3节概述我们的设计以及各实现层如何配合。第4 节详细介绍正则表达式层及其实现。第5 节介绍基于自动机的解析器及其 Moore 机类型化指令语言。第6 节评估该库：虽然空间和时间复杂度相对于被解析流的长度呈线性，但正则表达式字面量的大小及其结构会使解析器的大小呈非线性增长。我们通过一些具有区分性但病态的示例，衡量 TyRE 相较于 Idris 2 的解析器组合子库的性能。我们还介绍了在开发流编辑库 Sedris 时使用 TyRE 的经验。第7 节回顾了直接相关的工作。第8 节作出总结。

旁注。Idris 2 是一门相对较新的语言。为了让这份手稿内容完整而简洁，我们会在需要时说明 Idris 的语法和语义，格式如下。 ▨

2 库概述

TyRE 为带类型正则表达式提供了两种构造器：字符串字面量，它们隐式定义输出解析树的形状；以及组合子风格正则表达式构造器，可以精细控制解析树的形状。为了了解 TyRE 的字符串字面量如何使用，我们来看一个用于匹配*hh*:*mm*格式时间戳的正则表达式。

使用正则表达式有两种相关的方式：

- 匹配：确定字符串是否匹配正则表达式：`match ：`
  `TyRE 一个-> 字符串-> 布尔值`
- 解析：当字符串匹配正则表达式时，返回一个解析树：
  `parse ： TyRE a-> 字符串-> Maybe a`

翻译 t 结果解析树的类型，一个在两种情况下，都只与解析相关，而非匹配 .

[1] J´erˆome Vouillon 的 `ocaml-re` 可从以下地址获取：https://github.com/ocaml/ocaml-re。
[2] TyRE 可从以下地址获取：https://github.com/kasiamarek/tyre 。 [3] Sedris 可从以下地址获取：https://github.com/kasiamarek/sedris 。

旁注。Idris 要求为所有顶层定义提供类型声明，例如 `match` 和 `parse`。如果你正在阅读彩色版手稿，我们使用语义高亮来区分： 已定义的 函数或常量； 绑定/绑定 的变量出现位置； 类型构造器；以及 数据构造器 (未显示)。

以下代码定义了一个正则表达式字面量：一种对正则表达式进行编码的字符串，采用了 Unix 类系统中标准记法的一种变体：

```
                    -- (range from 0 to 9)
time1 : TyREUnit-- v v-(恰好匹配 `: ')time1 = r"(([01][0-9])|(
[2][0-3]):[0-5][0-9]"-- ^ ^ ^（表达式的交替）--| （字符之一 '0'和
'1'}）（从字符串字面量构造带类型正则表达式的智能构造器）
```

旁注。注释以双短横线（`--`）开头。

这些注释用 ASCII 艺术箭头（`^`,`v`）指出语法。正则表达式字面量会逐字匹配字符，其中少数关键字符具有特殊作用：

- 方括号（`[]`）列举备选项，可通过以下方式实现：– 穷举（`[01]`）；或 – 字符范围（`[0-9]`）。
- 竖线（`|`）表示两个子正则表达式之间的交替；以及
- 圆括号（`()`）有助于正则表达式字面量解析器 parse 该字面量。

要根据正则表达式验证选定的字符串，请使用`match`：

```
validateTimestamp : 字符串-> 布尔值
validateTimestampstr  = matchtime1str
```

图。1 包含一个 `TyRE` 用于将时间戳解析为结构更明确的格式：一对任意精度的整数值，分别表示小时和分钟。

旁注。Idris 对 `(c1、c2)` 和对类型 `(字符、字符)`与 Haskell 的一样，使用带括号的序列记法。 `TyRE` 类型构造器具有 函子的结构，提供了 `map` 函数（见下文）。中缀运算符二选一：由词法指定的运算符字符序列（`<*>`），或由反引号包围的任意函数名（`` `or` ``）。Idris 支持使用 **`where`**子句定义局部定义。

正则表达式字面量中的感叹号 (`!`) 是 TyRE 特有的特殊字符，其作用是带类型捕获组。我们将为匹配带有感叹号标记的子正则表达式的子字符串生成解析树，而其他子字符串将解析为`Unit` 值 `()`。正则表达式 `time1` 的类型恰好是 `TyRE Unit`，因为它没有捕获组。TyRE 库会解析字符串字面量并计算其形状：其解析树类型。与正则表达式字面量混合使用的智能构造器可以精细控制形状：

- 我们使用 TyRE 的函子 实例`TyRE`：

```
map : (a -> b) -> TyRE a -> TyRE b
```

```
--              v---------------v-（结果解析树的形状）
time2 : TyRE(整数,       整数)
time2 =
  --  (map : (a -> b) -> TyRE a -> TyRE b)
  --  v                      v-（TyRE 中表示捕获组的记法）
      map f (r "([01][0-9])!"          `or` r "([2][0-3])!")
  <*> map f (r ":([0-5][0-9])!") -- ^（或：TyRE a ->TyRE a ->TyRE a）
  --^((<*>) : TyRE a -> TyRE b -> TyRE (a,b))

  其中 digit : 字符 -> 整数digit c = 转换 c -
  转换 '0'f: (字符, 字符) -> 整数f(c1, c2)
   = 10 * digit c1 + digit c2
```

图 1：结构化时间戳解析器。

- 均匀交替的工作方式与交替相同，但两个子正则表达式具有相同的形状，并且结果不会跟踪匹配的是哪个备选正则表达式：

```
or : TyRE a -> TyRE a -> TyRE a
```

我们将正则表达式传递给 `parse`，它会返回二选一的结果：`Just` 如果字符串匹配该正则表达式，则返回一个由一对整数构成的解析树；或者返回`Nothing`：

```
parsedTimestamp : 字符串-> Maybe (整数, 整数)
parsedTimestampstr = parsetime2str
```

TyRE 还提供其他字符串解析函数：

- 前缀解析：根据贪婪标志解析最长或最短的匹配前缀，同时返回字符串中尚未解析的剩余部分：

```
parsePrefix : TyRE 一个->
                  字符串 -> (贪婪 : 布尔值) -> (Maybe a,     字符串)
```

- 不相交匹配：某些文本处理任务需要找出字符串中正则表达式的所有匹配项：

```
disjointMatches : (tyre : TyRE a) ->{auto 0 消耗 :
IsConsuming tyre} -> 字符串 ->(贪婪: 布尔值) -> (列表 (列表
字符, a), 列表 字符 )
```

给定的 `tyre` 必须是 消耗型的 — 只能匹配非空字符串，并且我们要求一个类型为`IsConsuming tyre`的（运行时擦除）见证。

旁注。 `auto` 参数用大括号（{}）括起来，表示 Idris 将通过一个称为证明搜索的过程来搜索此参数。它会遵循用户定义的提示，尝试可见的数据构造器，并优先选择类型合适的绑定变量。Idris 通过将实例解析脱糖为证明搜索来实现类型类。

• 替换：替换字符串中所有找到的不相交匹配项：substitute : (tyre: TyRE a) ->{auto 0 consuming : IsConsuming tyre} -> (a -> 字符串) ->字符串 -> 字符串例如，我们更改某段文本中时间的显示格式。changeTimeFormat : 字符串 -> 字符串 changeTimeFormat = substitute time2 (\case (h, m) =>show m ++ " past " ++ show h) -- ^-（模式匹配匿名函数）

旁注。关键字 \case 表示一个匿名函数，它会立即对其参数进行匹配。

调用 changeTimeFormat "看，它是 11:15。"的计算结果为 "看，现在是 11 点过 15 分。"。

• 流： TyRE 库不仅可用于解析字符串，也可通过以下函数对 字符流进行词法切分：

```
getToken : (贪婪: 布尔值) -> TyRE a-> 字符流 ->(Maybe a,
字符流)
```

旁注。Idris 流是无限的惰性列表。下面是一个简单的词法分析器，用于对由数字构成、以分号分隔的流进行词法切分：getDigits :字符流 -> 流 (Maybe Integer)getDigits = -- v-(match : 字符 -> TyRE())unfoldr(getToken True (match ';' *> 数字))--((*>) :TyRE a -> TyRE b-> TyRE b)-^ ^(digit: TyREInteger)

我们使用了三个智能构造器：

– 精确匹配所选字符：match : 字符-> TyRE ()– 数字 TyRE：digit : TyREInteger– 一种正则表达式组合子，它会丢弃第一个解析树，并且只返回第二个：(*>) : TyRE a-> TyRE b-> TyRE b

3 架构

TyRE 由两个主要部分组成：

• 一个正则表达式层，其中包含带类型正则表达式和面向用户的接口：

– 智能构造器；– 具有计算/推断形状的正则表达式，我们称之为无类型正则表达式；以及– 正则表达式字面量。

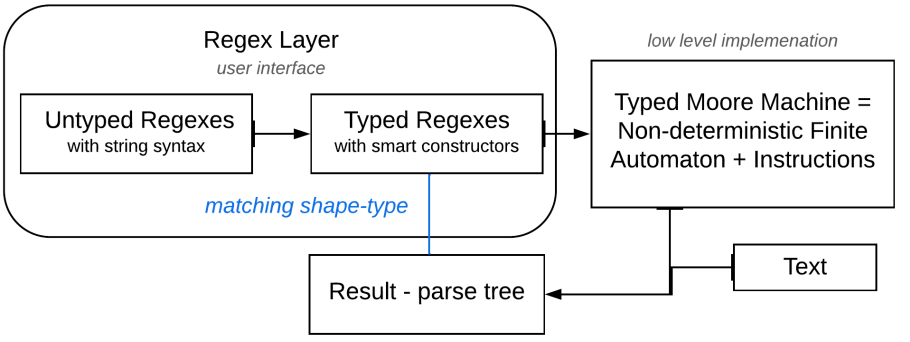


图 2：解析器架构概览。

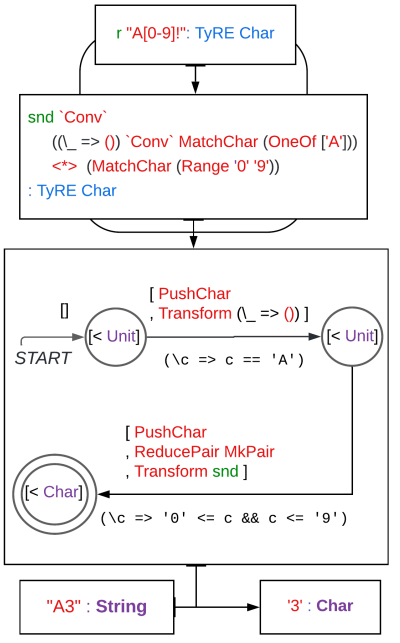
- 一种类型化的 Moore 机，在为正则表达式执行非确定有限状态自动机的同时构建解析树。

图 2 展示了 TyRE 的架构。用户可以直接构造带类型或无类型正则表达式，也可以将其写成正则表达式字面量；正则表达式层会将它们转换为核心带类型正则表达式 Rep 表示。随后，通过 Thompson 构造的一个变体生成带类型 Moore 机，即一种非确定性有限状态自动机，其转移还会在低级 例程带类型语言中发出动作。给定输入文本，我们执行此 Moore 机，要么报告正则表达式 match 失败，要么返回结果解析树。

图 3 展示了各组件之间的联系及其职责。我们使用 `"A3"` 来解析字符串 正则表达式 `"A[0-9]!"`。TyRE 首先将正则表达式字符串字面量解析为带类型正则表达式 r_0: TyRE 字符。随后，我们使用 Thompson 构造的一种变体（McNaughton 和 Yamada，1960年；Thompson，1968年）构建带类型 Moore 机。图 3a 展示了得到的带类型机器。它是正则表达式字面量 `"A[0-9]"`对应的非确定有限自动机：箭头下方的标签是一个 字符谓词，用于确定何时执行此转移。箭头上方的标签是要在每次转移时执行的代码例程，当我们到达接受状态时，它们将生成正确的解析树。

我们将选定的字符串 `"A3"`输入 Moore 机。遍历非确定有限自动机时，我们会在栈上收集解析树的各个部分[4]。图 3b 记录了这台 Moore 机执行期间的栈状态，将当前的非确定有限自动机状态——当前读取的字符以及用于守卫下一次转移的条件——与针对该转移执行的 Moore 机代码及其对辅助栈的影响交错呈现。为确保栈中的内容能够生成格式良好的解析树，我们为每个状态关联一个栈形状。它描述了此时栈必须具备的形状。具体而言，栈形状是一个类型序列，用于描述解析树结果中哪些部分已经在栈上构造完成。我们用一个由类型构成的 `SnocList` 来实现栈形状。每条代码例程指令都根据其执行前后的假定栈形状来赋予类型，因此其类型判断是一种霍尔三元组。我们为唯一接受状态关联的栈形状是单元素 `[<字符]`： `SnocList Type`，与输入形状 `TyRE`

[4] 尽管 Moore 机器在其辅助状态中使用 Stack，但该 Stack 不会影响执行的控制流，控制流完全由当前状态和当前字符决定。因此，计算模型仍然是正则非确定有限状态自动机，而不是下推自动机。



例程	Stack
	[<]
当前字符: 'A'. 条件检查: 'A' == 'A'	
PushChar 变换(\ =>())	[<'A'] [< ()]
当前字符: '3'. 条件检查: '0' <= '3' && '3' <= '9'	
Push字符 归约对 MkPair 转换 snd	[< (), '3'] <escape>[<</escape>(), '3'<escape>]</escape> [< '3']
接受。	

（a）翻译　　(b) 执行堆栈跟踪

图 3：使用正则表达式字面量 "A[0-9]!" 解析 "A3"。

字符。因此，沿 Moore 机成功运行的路径保证我们能够提取出形状正确的解析树。

旁注。 Idris 支持以下用于左嵌套列表的语法糖，其中：

`[< x1, x2, ..., xn]` desugars into `(((Lin :< x1) :< x2) ··· ) :< xn` ▨

4 正则表达式层

正则表达式层提供面向用户的抽象，接下来我们将详细介绍。当用户使用智能构造器 `r`，

- 正则表达式字面量 在编译时被解析为一个
- 无类型正则表达式（正则表达式），其推断形状也会在编译时计算，然后转换为一个
  - 带类型正则表达式 (TyRE)。

4.1 正则表达式字面量

TyRE 提供字符串字面量，用于创建类型化正则表达式。借助 Idris 的类型级计算，我们在编译时将字符串字面量 parse 为类型化正则表达式。这项功能既能确保字符串是有效的正则表达式，也能计算结果解析树的类型。我们分两步 parse 正则表达式字面量。首先，使用解析器组合子库将字符串字面量 parse 为无类型的正则表达式数据结构（正则表达式）[5]。第二步，我们利用解析树与形状简化规则之间的一对一映射，将无类型正则表达式转换为带类型正则表达式。

表1展示了字符串正则表达式与正则表达式之间的映射关系，并定义了对预期无类型正则表达式形状的一种近似表示。然而，实际正则表达式的形状计算要复杂得多：

- 每个未被包裹在由构造器包裹的子正则表达式，在正则表达式字面量中由感叹号（`!`）标记，都会被扁平化为一个`Unit`，例如：

  `r "((([a-z])+)!)|(hj)"` 的形状为二选一 (列表 字符) `Unit`

- 我们消除冗余`Unit` 类型（types）在保留的子正则表达式中。具体而言，我们简化：

  – 一对`Unit`到一个 `Unit`；– 一个 列表 Unit到一个 `Nat`，表示列表的长度；– 一个 二选一 Unit Unit 到一个布尔值值。
  例如：

  `r"((([a-z])+)!)|(hj)"` 形状为字符列表`r"((ab*[vkw]([a-z])+)|(hj))!"` 形状为`Maybe` (自然数, (字符, 字符列表))

要在编译时 parse 正则表达式字面量，我们使用 Idris 标准库中解析器组合子库的一个副本。在我们复制的解析器组合子库中，所有函数都限定为 **`public export`**。因此，类型检查器会将格式正确的正则表达式字面量求值为 正则表达式 类型的值。

旁注。Idris 的定义带有可见性限定符：

- **`export`**：客户端模块可以使用该定义及其关联类型；
- **`public export`**：客户端模块的类型也可以依赖此定义的函数体；
- **私有** （默认）：客户端模块不得直接引用此定义。 ▨

4.2 类型化正则表达式

类型化正则表达式会明确表达解析树的形状。对于类型为 `TyRE a` 的带类型正则表达式，解析的结果将是类型为 `a` 的解析树，或什么也没有（`Maybe a`）。图 5 定义了`TyRE`数据结构。它有两类构造器。

[5] 格式正确的正则表达式字面量所构成的形式语言是上下文无关语言，而非正则语言，因此我们需要解析器组合子所提供的额外表达能力。

字符串正则表达式	RE	形状	保持形状 (KS)
r "a"	恰好 一个	Unit	Unit
r "[ab]"	OneOf ['a' , 'b']	Unit	Char
r "[a-c]"	To 'a' 'c'	Unit	Char
r "."	Any	Unit	Char
r "AB"	Concat A B	(形状 A, 形状 B)	(KS A, KS B)
r "A|B"	交替 A B	二选一 (形状A) (形状 B)	二选一(KS A) (KS B)
r "A?"	Maybe 一	Maybe (形状A)	<escape>Maybe (KSA)</escape>
r "A*"	零次重复 A	列表 (形状A)	列表(KSA)
r "A+"	Rep1 A	列表 (形状A)	列表 (KSA)
r "A! "	保留 A	KS A	KS A

表 1：正则表达式字面量语法，正则表达式 及其关联形状

数据字符条件 **：** 类型 **其中**||| 如果字符包含在集合中则匹配 OneOf **：** 有序集合 字符 -> 字符条件||| 如果字符在范围内（含边界）则匹配Range **：** (字符, 字符) -> 字符条件||| 如果字符匹配谓词则匹配Pred **：** (字符-> 布尔值) -> 字符条件

图 4：单字符条件。

第一类是常见的Kleene代数运算：空字符串正则表达式、条件字符正则表达式——图 4定义了单字符条件、连接、交替和重复的运算。没有用于表示空语言的构造器。我们可以通过向 字符谓词构造器传入一个不可能成立的谓词来表示空语言，例如 字符谓词 （谓词 （\ =>False））。

第二类构造器在不改变正则表达式的情况下操作解析树形状。 Conversion 构造器根据给定的函数转换解析树。正是这一构造器使我们能够实现第 Functor TyRE 节中提到的实例。 2。 Group 构造器会忽略解析树形状，直接返回匹配字符的子串。通过放弃结构化解析，我们将正则表达式解析替换为正则表达式匹配。匹配使得自动机最小化技术得以应用，从而实现更快速的解析器。

5 Moore 机

我们通过执行带类型的摩尔机进行 parse。类型化机器代码维持了语义不变量，否则可能需要大量证明才能保证这些不变量（Morrisett 等人，1999 年）。在依赖类型语言中，嵌入此类

data TyRE : 类型 -> Type where| | | 空词（ε）的正则表达式 Empty :TyRE () | | | 单字符正则表达式 MatchChar :字符条件 -> TyRE Char| | | 连接。当一个词可以拆分为| | | 两个连续部分，且它们按顺序匹配子正则表达式时，该词即匹配。 (<*>) :TyRE a -> TyRE b -> TyRE (a, b)| | | 交替。当一个词匹配任一子正则表达式时，该词即匹配。 (<|>) : TyRE a ->TyRE b -> TyRE (二选一 a b)| | | 克莱尼星号。当一个词可以拆分为| | | n 个连续部分，且每部分都匹配参数时，该词即匹配。 Rep :TyRE a -> TyRE (SnocList a)| | | 形状变换。| | | 保留底层正则表达式，并根据给定函数转换解析树的结果。| | | Conv :TyRE a -> (a -> b) -> TyRE b| | |放弃结构化解析并返回匹配的子串。| | | 生成的解析器只收集已消耗的字符。可能| | |生成更小（=更快）的解析器。 Group :TyRE a -> TyRE 字符串

图 5：类型化正则表达式

类型化机器语言（McKinna 和 Wright，2006）。我们将机器划分为两个概念上截然不同的部分：

- 非确定有限自动机： 执行匹配的非确定有限自动机；和
- 例程：对非确定有限自动机（NFA）的转移进行标记的指令列表，沿着一条接受路径执行这些指令，保证能够构造出一个有效的解析树。

NFA 组件控制 Moore 机的控制流，其例程组件则控制机器的累积副作用。

5.1 抽象化

Moore 机层操作若干关键的编程抽象。本小节将介绍这些抽象，以及在 §5.3 中实现它们的数据类型。

非确定有限自动机。一个定义在输入符号集合 （称为字母表）上的非确定有限状态自动机是一个元组（Q，S，q_{acc}，$next$，P），由以下部分组成：

- Q：一个有限的状态集合；
- $S \subseteq Q$：一组起始/初始状态；
- acc $\in Q$：一个特殊的接受状态；

- 下一个：$(Q \setminus \{\text{acc}\}) \times \Sigma \to \mathcal{P}Q$：一种关系，即转移关系。
  （我们用 $\mathcal{P}$表示幂集。）

每个$q_2 \in next(q_1, a)$ 表示一条转移 $q_1 \xrightarrow{a} q_2$ 从状态 q_1 到 q_2 由符号 a标记。在这种表述中，接受状态没有出边，并且我们不允许静默转移（通常称为 ε 转移）。这种选择不会影响我们能够表达的解析器类别。我们只考虑基于字符的自动机。

运行时模型。遍历非确定有限自动机（NFA）时，我们会构建解析树的各个部分，并将它们收集到异构栈中。更准确地说：

- 为确保接受时栈中的值构成有效的解析树，我们根据栈形状 为类型建立索引——栈形状是一个左嵌套的类型列表，每个栈位置对应一个类型：形状：类型形状 = SnocList 类型

- 我们执行一个线程池，使线程在非确定有限自动机的状态间转移：`ThreadData` :形状 -> 类型
- 每个线程数据都可以记录当前 match 的子串，并维护一个存放部分构造的解析树的栈：`Stack` :形状 -> 类型

例程。例程是由指令组成的列表，每条指令都会影响线程数据。为了确保指令生成正确类型的解析树，我们为每条指令编制索引，因此也为每个例程编制索引，依据的是执行前的栈形状这一假设，以及执行后所保证的栈形状：

```
指令：(pre, post：形状) ->类型^ ^ -- ---------- --（执行指令/例程前后的
栈形状）-- v----------v例程：(pre, post：形状) ->类型
```

有一条指令——将最近读取的符号压入栈中——在机器尚未读取任何符号之前无效。因此，我们还要标明机器初始化时可以执行哪些指令：

```
初始化例程：例程xs ys-> 类型
```

类型化 Moore 机。由于我们的非确定有限自动机包含一个独特的接受状态，我们将使用`Maybe` 类型构造器来实现 Moore 机的状态，其中由`Nothing` 表示该接受状态。我们将根据结果解析树类型为 Moore 机编制索引：

```
摩尔机 : (t：类型) -> 类型
```

每个摩尔机都会为每个非接受状态指定预期的栈形状：

```
查找 : s -> 形状
```

以及 s o 连同结果解析树类型t，我们为每个状态关联一个栈形状 :

```
shapeOf : (lookup : s ->形状) -> (t : 类型) ->Maybe s-> 形
状 shapeOf lookup t (Just s) =lookup s -- 非接受状态shapeOf
lookup t Nothing = [< t ] -- 接受状态
```

5.2 旁注：*Idris* 预备知识

我们还需要了解 Idris 的一些其他特性，以便讨论 Moore 机层的实现。

数量。Idris 2（Brady，2021）的一项重大创新是其数量注释，遵循 McBride（未发表）和 Atkey（2018）的定量类型论。例如，我们可以定义以下半擦除和类型：

```
数据 EitherErased : 类型 -> 类型 -> 类型 其中Left :
(x : a) ->EitherErased a bRight : (0y : b) ->
EitherErased a b
```

它的 `Left` 构造器与更为熟悉的标准 二选一 类型构造器相似。不过，它的 `Right` 构造器的参数标注了数量**0**，这表示该参数会在运行时被擦除 。因此，`Right y` 在运行时表示为一个标签，这意味着 `EitherErased a b`的值与类型为 `Maybe a`的值具有相同的运行时表示。Idris 2 支持以下数量：

- 运行时擦除（**0**）；
- 线性（**1**）：变量在运行时必须且只能使用一次；以及
- 任意（未标注，默认值）：变量可以使用任意次数。

记录。Idris 2 的记录 会被反糖化为仅含一个构造器的**数据** 声明，并为每个字段生成投影函数。下面是依赖对的定义示例：

```
记录 DPair a (b : a-> 类型) where
构造器 MkDPairfst: asnd: b fst
```

它会被反糖化[6] 为数据声明：

[6] Agda 行家可能会期待记录支持 η-相等，也就是判断相等：`rec = (MkDPair rec.fst rec.snd)` 由于普通的**数据**声明在判断层面不支持这种 η-相等，Agda 记录无法简单地反糖化为单构造器**数据**声明。Idris 将来或许会支持记录的 η-相等，但目前尚未支持，因此会以相应方式反糖化记录。

```
数据DPair : (a: 类型) -> (b : a-> 类型) -> 类型其中
MkDPair : (fst :a) -> (snd : b fst) -> DPair a b
```

以及两个称为投影的定义，它们使用 Idris 的后缀记法：

```
(.fst) : DPair a b ->a(MkDPair x
y).fst = x(.snd) : (rec: DPair a b) -
> b rec.fst(MkDPair x y).snd = y
```

Idris 为依赖对提供了特殊语法，其类型为：

```
(shape :形状 ** Stack shape)
```

会脱糖为依赖对类型：

```
DPair形状 (\shape  => Stack shape)
```

以及 形状和一个形状适当的 Stack：

```
([< 自然数, 布尔值] **  [< 42, 真])
```

展开为依赖对：

```
MkDPair  [<自然数, 布尔值] [< 42, 真]
```

Record 字段可以包含数量注解，例如，Idris 的标准库中包含一个字段在运行时被擦除的记录：

```
记录 Subset a (b :a -> 类型) 其中构造
器 Elementfst: a0 snd: b fst
```

我们将使用这个依赖对，通过要求依赖对的（.snd）字段包含一份证明，表明该值符合要求，从给定类型中排除某些值。例如，表示初始化安全指令的类型 IsInitInstruction。由于该字段会在运行时被擦除，我们只会传递指令，而不会传递其安全性证明。

隐式参数。Idris 支持两种隐式参数，并会尝试在 elaboration 过程中构造它们：

- 普通的 隐式参数将使用其动态模式合一算法进行推导（Miller，1992；Reed，2009；Gundry，2013）。我们使用花括号声明它们。例如，稍后我们将把线程的运行时状态定义为以下 Record，即其非确定有限自动机状态及相关的ThreadData：

```
Record 线程 {t :类型} (machine : 摩尔机 t) where 构造器
MkThread
  状态 : Maybe machine.s
  tddata : ThreadData(shapeOf machine.lookup t 状态)
```

用户只需提及 线程 machine，因为 elaborator 可以根据给定的 Moore 机 t 的类型推断出预期的结果解析树类型 machine。Idris 默认采用两种隐式参数约定：

– 关键字 **forall** a. 会绑定一个名为 a 、数量为 **0** 的隐式参数，并尝试使用合一推断其类型；– 类型声明中未绑定的隐式参数，会在声明开头自动绑定为数量为 **0** 的隐式参数。此功能可以关闭，它能够使类型声明更加简洁，类似于前束多态语言中的类型声明。

- 证明搜索隐式参数除了通过合一进行推导外，还将遵循用户声明的统一提示（Asperti 等人，2009），尝试使用数据/类型构造器和记录投影，并填入绑定变量。我们使用**auto** 关键字来声明证明搜索隐式参数，正如我们在第4页提到的那样。例如，稍后我们将把 Moore 机定义为带有一个证明搜索隐式参数的记录：

```
记录 摩尔机 (形状 :类型) 其中构造器
MkMooreMachine0s : 类型0 查询 :s -> 形状{
auto isEq: 相等 s}初始状态: 初始状态类型
形状 s 查询下一状态: 转移关系 形状 s 查询
```

类型 相等 s 是一个记录，其中有一个字段，用于表示机器状态类型上的相等谓词s：

```
(==) : s -> s -> 布尔值
```

通过为记录的构造器指定一个证明搜索参数， elaborator 将搜索 Eq t 的值，以使用证明搜索隐式参数的方式表达类型类解析（Sozeau 和 Oury，2008）。

5.3 实现数据类型

现在，我们可以定义用于在 TyRE 中表示摩尔机的数据结构。

栈。我们使用由其形状索引的异构栈：

```
数据Stack : 形状 -> 类型其中Lin : Stack [<](:<) :
Stack tps -> (e : t) -> Stack (tps :< t)
```

```
数据 Instruction : (pre,post : 形状) -> 类型 其中||| 将选定的符
号压入栈中 Push : {0 x: 类型} -> x-> Instruction tps (tps :<
x)||| 将当前已消耗的字符压入栈中 PushChar : Instruction t
ps (tps :< 字符)|||根据指定的函数归约栈顶的两个符号|||
ReducePair : {0 x, y, z: 类型} -> (x -> y -> z) -> Instruction
(tps :< x :< y) (tps :< z)||| 映射栈顶元素 Transform : {0x,y :
类型} -> (x -> y) -> Instruction (tps :< x) (tps :< y)||| 将收集
到的字符串压入栈中 EmitString : Instruction tps (tps :< 字符串)
||| 设置记录标志并开始收集字符 Record : Instruction tps tps
```

图 6：Moore 机指令集

因此，Stack 要么为空，要么由一个给定形状的 Stack（tps）和其顶部的一个值组成。

旁注。Idris 对 snoc 列表记法的去糖约定纯粹是句法上的[7]，因此我们可以使用 snoc 列表记法，隐式地将任意构造器或函数调用的序列向左嵌套Lin 和 snoc (:<)。▨

线程局部状态。每个线程的状态由一个根据线程数据的形状索引的栈以及一个位组成，rec 用于确定该线程是否需要record当前解析的子串。在这种情况下，更新专门用于存储字符的字段已记录的内容，直到目前为止：

```
Record ThreadData(形状 : Shape)
where 构造器 MkThreadDatastack :Stack
形状已记录内容 :SnocList Charrec: 布尔
值
```

指令集。图 6 展示了 Moore 机指令，即可能出现在非确定有限自动机转移例程标签中的指令。它们会作用于相应的 ThreadData 记录。在遍历 Moore 机时，我们通常会将有关解析树的信息收集到栈上。此规则有一个例外。执行 Moore 机时，我们可以通过提高 record

[7] 如果你正在阅读彩色版手稿，不妨试试这个谜题：构造下面的项，使每个字符都具有指定的语义角色：[n,m,n]，也就是说：依次为类型构造器、绑定变量、数据构造器、定义、类型构造器、绑定变量和数据构造器。

标志进入特殊模式，这通过 Record 指令实现。当 record 标志升起时，我们还会将已消耗的字符记录到专用缓冲区 recorded中。可以使用 recorded 指令将这些字符压入栈中；该指令还会降低 EmitString 指令，同时关闭 record 标志，使机器恢复到非记录模式，并清空缓冲区。创建 Group 子正则表达式的解析树时，我们会使用记录模式。其他所有指令都会通过压入栈或对栈进行变换来改变栈。我们定义谓词IsInitInstruction，使其包含除Push以外的所有指令。

栈形状 pre 和 post 为每条指令建立索引的栈形状依赖于线程的栈在执行前具有适当的形状，并保证执行后具有给定的形状。然后，我们可以通过实现以下类型的函数，在具有适当形状的栈上执行一条指令：

```
execInstruction : (r: 指令 pre post) ->二选一抹除 字符
(IsInitInstruction r) ->ThreadData (p ++pre) ->ThreadData
(p ++post)
```

它需要二选一：要么是上次读取的字符，要么是运行时擦除证明，证明该指令不需要此字符。随后，它将接收一个 ThreadData，其栈的顶层符合指令栈形状的前置条件，并生成一个 ThreadData，保证满足后置条件。

我们用例程标记 Moore 机的转移：由指令构成的 cons 列表，在其索引中以望远镜式方式折叠中间的栈形状。

```
数据 例程 : (前置,后置 :形状) -> 类型 其中||| 空例程空
值构造器 :例程 形状 形状||| 向例程前置一条指令(::) :
指令 前置 中间 -> 例程 中间 后置->例程 前置 后置
```

将执行从指令扩展到例程是很直接的，只需采用归纳方式。

带类型 Moore 机。为了表示 §5.1 中的 NFA 抽象，同时简化类型安全性证明，我们将 NFA 中的每个转移与其标记例程配对。我们将用列表表示出边集合，以此表示转移关系，如下所示：

```
转移关系:  (形状: 类型) ->(状态 : 类型) -> (状态形状 : 状态 -> 形
状) -> 类型 转移关系 形状 状态 状态形状=(st : 状态) ->(c : 字符) -
> 列表 (st' : Maybe 状态 ** 例程 (状态形状 st)
```

```
                                    (shapeOf状态形状 形状 st       '))
```

```
记录 摩尔机 (形状 : 类型) 其中构造器
MkMooreMachine0 s: 类型0 查找 : s-> 形状{
auto 相等 :相等 s}初始 :初始状态类型 形状 s
查找下一个 :转移关系 形状 s 查找
```

图 7：TyRE 的 Moore 机类型

因此，我们通过以下方式对转移关系的类型进行参数化：

- 形状：结果解析树类型；
- 状态：非接受NFA状态的类型；以及
- 状态形状：每个状态所假定的栈形状。

给定这些参数、一个转移关系和给定输入

- st：起始状态；以及
- c：字符acter

将它们映射为一个列表，该列表表示一个有限的转移集合$st \xrightarrow{r\ c} st'$：

- st'：目标状态，可能是接受状态Nothing；以及
- r：此转移的标记例程。它假定与 st 关联的栈形状，并保证给定 shapeOf（参见第 12 页）所定义的、与 st' 关联的栈形状：– 当 st' 是非接受状态 Just s' 时，它是 stateShape s'。– 当 st' 是状态 Nothing 时，它就是给定的形状。

同样地，如果我们将起始状态视为无源转移的目标状态，那么我们将一个以列表表示的初始状态 Subset 表示如下：

```
初始状态类型 : (形状 : 类型) ->(状态 : 类型) -> (状态形状 :状态 -> 形状)
-> 类型 初始状态类型 形状 状态 状态形状 = 列表 (st :Maybe 状态 **
Subset (例程 [<] (shapeOf 状态形状 形状 st)) 初始化例程)
```

因此，我们将每个初始状态 Start $\xrightarrow{r}$ st 表示为：

- st：一个状态，可能是接受状态Nothing；以及
- r：一个初始化例程，因此我们需要一个运行时擦除证明，证明该例程仅包含初始化安全指令。

图 7 展示了 TyRE 中穆尔机的记录声明。其字段：

- s：非接受状态的一种运行时擦除类型

- `lookup`：为每个非接受状态关联一个形状的运行时擦除结构；
- `isEq`：非接受状态上的隐式相等谓词，使用 Idris 的证明搜索机制进行 elaboration；
- `init`：初始状态及其初始化例程；
- `next`：标记有解析例程的转移关系；

我们在两个地方使用相等谓词`isEq`：

- 在自动机最小化过程中；以及
- 在执行过程中，为了维持线程数量有界。具体而言，如果两个线程处于相同的状态，这意味着它们都已 parse 相同的子字符串，并且在后续执行中拥有相同的可用路径。它们之间唯一可能的区别在于迄今为止构建的解析树部分。处于相同状态意味着迄今为止构建的解析树必须具有相同的形状。由于我们的目标并非构建所有可能的解析树，而是返回其中一种可能结果，因此我们只保留这两个线程中的一个继续执行。这样可以确保线程数量不超过自动机中的状态数量，从而保证线性时间复杂度和空间复杂度。

5.4 Thompson 构造

McNaughton 和 Yamada（1960年）和 Thompson（1968年）描述了将正则表达式转换为等价自动机的算法。我们的编译方案是对这些著名构造的标准变体，并加入了代码例程标签来执行解析。编译 `Group` 子正则表达式时，我们会使用一种单独的机制来为包装在组中的正则表达式构建 Moore 机。图 8展示了如何为除 `TyRE` 之外的所有构造器构建状态机：`Group`。我们将在 §5.5中介绍这个剩余情形。

图 8a：谓词。接受单字符词语言的状态机具有：

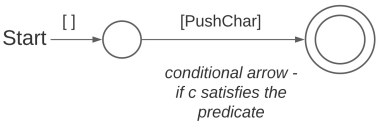
- 一个具有空初始指令列表的单一起始状态；
- 由给定谓词进行条件约束的、通往接受状态的单个转移；以及
<escape>• </escape>一条标记此转移的指令：`PushChar`。

我们通过定义每个字符条件何时成立，统一采用这种构造：

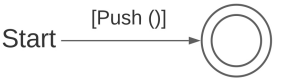
```
satisfies : CharCond -> Char -> Bool
satisfies (OneOf xs    ) = \c => contains c xs
satisfies (Range (x, y)) = \c => x <= c && c <= y
satisfies (Pred f      ) = \c => f c
```

图 8b：空字符串。在空字符串 ε的状态机中，接受状态同时也是起始状态，且是唯一存在的状态。它只有一条初始化指令：`Push ()`。

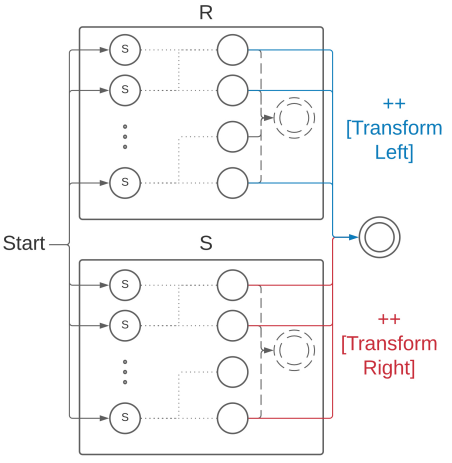
图 8c：交替。要为 $R|S$ (交替 `R S` 构建状态机，我们首先分别为 R_{m} 和 S_{m} 构建状态机。交替状态机的起始状态是 R 和 S 的起始状态的并集。我们保留所有转移，并将接受状态合并为一个，因此要到达该接受状态



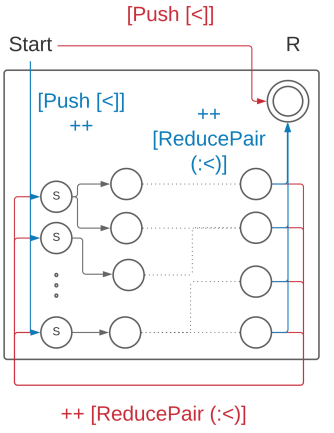
(a) 对单个字符的谓词。



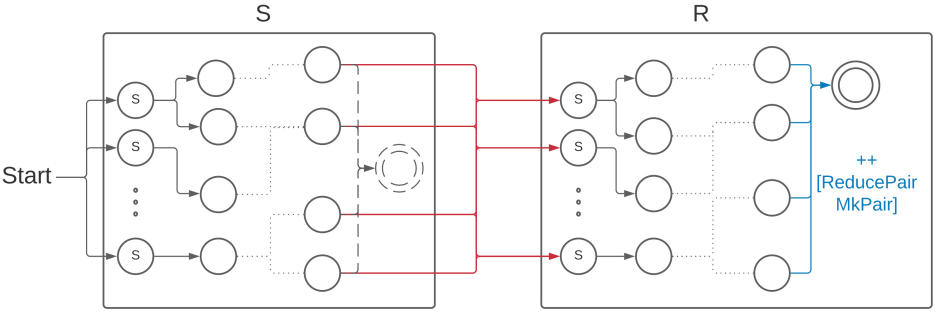
(b) ε — 匹配空字符串



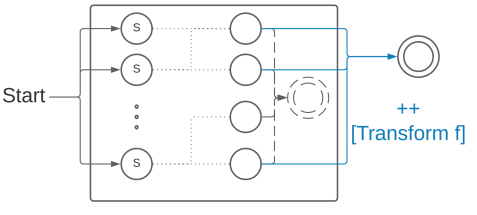
(c)**R|S** — 匹配 R 或 S



(d) **R*** — 匹配零个或多个 R



(e) **RS**— 匹配 R 后接 S



(f) Conv f**R** — 通过形状变换匹配 R

图 8：**Moore 机的构造**。

我们遍历二选一的 R_m 或 S_m。我们会跟踪已遍历的是哪一个，并将收集到的解析树提升为从`Either` 类型到从R_m 到接受状态的所有转移所需的类型。为此，我们在例程末尾添加一个`Transform Left`指令；同样地，对于从S_m 到接受状态的转移，添加一个`TransformRigth` 指令。

图 8d：克莱尼星号。我们首先通过构建R^* (`Star R`) 的状态机来构建状态机R_m 用于R。新状态机中的起始状态来自 R_m 的起始状态以及接受状态。我们在每个初始化例程前添加一个`Push [<]` 指令，将一个累积用的空`SnocList` 压入 Stack。因此，如果我们直接到达接受状态，解析树将是空的`SnocList`。我们保留前缀解析器R_m 中的所有转移。对于每个进入接受状态的转移，我们：

- 添加一个返回起始状态的新转移（如果需要）；并且
- 对于现有转移和新转移，都向例程追加一个 `ReducePair (:<)` 指令，将最新的 R解析树压入累积的`SnocList`。

图 8e：连接。要为 RS（Concat R S）构建状态机，我们首先分别为 R_m 和 S_m 构建状态机。连接运算的状态机中的初始状态将是 R_m中的初始状态。我们保留 R_m 中的所有转移，但会导向接受状态的转移除外。对于后者中的每个转移，我们还会向 S_m中的每个初始状态添加新转移，同时保持所有 S_m 不变。这样，我们现在会先遍历 R_m，然后遍历 S_m，最终 Stack 顶部将包含 R 的解析树和 S 的解析树。我们通过在每个例程标签前置一个 ReducePair MkPair 指令，将二者组合起来。

图 8f：转换。转换操作（`Conv R f`）的状态机就是 R的状态机，并在每个接受转移后追加一个`Transform f` 指令。

一项优化。通常，在递归构造 Moore 机时，我们会将一条指令添加到现有例程的最末端。这发生在 `<|>`、`<*>`、`Rep`和`Conv`中。另一方面，我们仅在 `Rep` （初始 `[Push [<]]`）中将元素添加到例程的开头。向列表末尾添加元素需要遍历整个列表，这并非最优。出于这一原因，我们在构造 Moore 机时，实际上会对例程使用 snoc-list 累加器，确保其组成指令的前置和后置栈形状保持一致，然后将列表反转为用于执行的例程。

5.5 组最小化

对于带类型正则表达式 `Group a`，解析树的形状是一个字符串。相应的状态机于是成为正则表达式匹配器，而不是解析器。我们利用这一性质，为组构建状态机创建了一个完全独立的机制。首先，我们构建一个非确定有限自动机（NFA）；然后执行一些优化以缩小 NFA；最后从 NFA 构建一个`MooreMachine String`。

在确定性情形中，自动机最小化可以高效地解决（Hopcroft，1971）。然而，在非确定性情形中，Jiang 和 Ravikumar（1993）证明了

NFA 最小化很困难。具体来说，他们证明了它是 PSPACE-完全的。因此，我们使用一些简单的启发式方法来最小化 NFA，并将实现最优最小化算法的任务留待未来工作，例如 Kamedaand Weiner的算法（1970年）。

我们使用 Thompson 构造的一种变体来构建适用于我们这类正则表达式的 NFA。这与为其他类型化正则表达式构建 Moore 机时使用的构造相同，但不包含例程。这里我们使用专门的数据结构来表示 NFA，其中包含初始状态列表，以及一个映射：对于每个状态，都映射到一个由以下内容组成的对列表：

- 自动机中的一个相邻状态；以及
- 一个字符条件字符谓词，用于约束到该相邻状态的转移。

这是对普通 NFA 的一种简化：按目标状态对转移关系进行分组。

优化。我们通过合并具有相同出转移的状态来缩小 NFA 的规模，也就是说，在相同条件下，这些状态会得到相同的后继状态集合。我们重复此操作，直到没有更多状态可以合并。我们通过比较构造器及其参数来比较条件。请注意，对于 `OneOf` 和 `Range`，我们只比较参数；但我们无法比较传递给 `Pred`的函数，因此始终假定它们各不相同。未来可以使用更高效的数据结构（例如二元决策图）来优化这一选择（Lee，1959年）。

从非确定有限自动机构建 Moore 机。我们从非确定有限自动机构建一个状态机。除接受状态外，每个状态的形状都是 `[<]`。我们定义以下代码例程：

- 我们使用例程初始化每个起始状态： `[Record]`
- 我们使用 `[EmitString]`标记所有指向接受状态的转移。
- 如果接受状态同时也是起始状态，我们使用 `[Record, EmitString]`对其进行初始化。
- 我们使用空例程 `[]`标记所有其他转移。

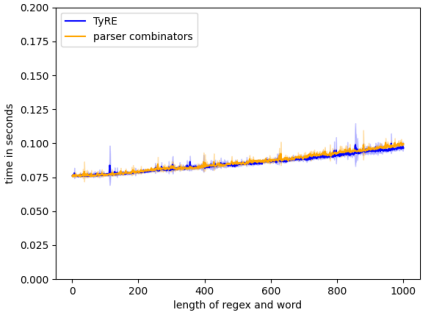
6 评估

我们通过两种方式评估 TyRE 库。首先，我们进行定量比较，将其与现有的解析器组合子库进行对比，并考察 TyRE 的设计是否以及如何提升 Idris 2 生态系统中的最先进水平。我们使用病态测试，着重考察 TyRE 与 Idris 2 解析器组合子库之间的性能权衡。第二种评估是定性的，报告我们将 TyRE 用于一个更大型应用（流编辑库）时的体验。

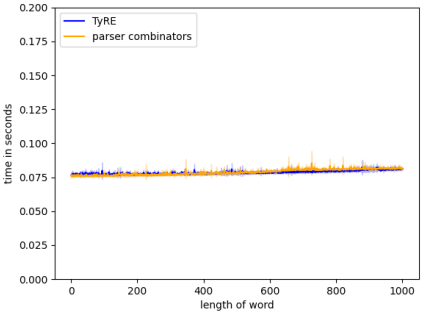
6.1 将 *TyRE* 与 *Idris 2* 解析器组合子进行比较

由于 Idris 2 没有现成的正则表达式解析器库，我们通过将 TyRE 与 Idris 2 的标准解析器组合子库进行比较来评估 TyRE。Idris 2 的标准解析器组合子库可以 parse 上下文无关语言，因此功能比正则表达式解析器更强大，不过并不具备相同的运行时复杂度保证。

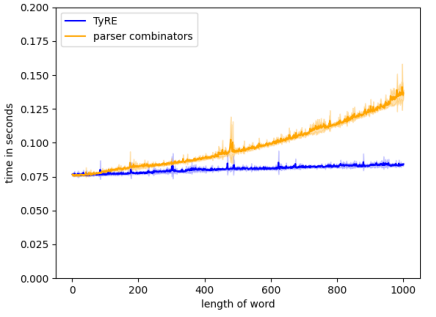
我们比较了 TyRE 和 Idris 2 标准解析器组合子库在四个病态示例中，对等价正则表达式解析器处理相同输入时的运行时间。



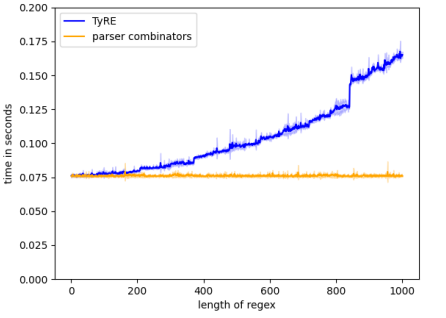
(a) 正则表达式：a^n；字符串：a^n



(b) 正则表达式：$a*$；字符串：a^n



(c) 正则表达式：$((a*c)|a)*b$；字符串：a^nb



(d) 正则表达式：$a(|a)^{n-1}$；字符串：a

图 9：TyRE 与 Idris 2 标准库解析器组合子库的解析耗时。

这些病态示例展示了两个库各自的优势与弱点。请注意，在沿水平方向增加正则表达式大小的示例中，生成的非确定有限自动机也会变大，而 TyRE 的运行时间会呈非线性增长。对于固定的 正则表达式，执行时间保证与所解析字符串的长度呈线性关系。

我们运行了[8] Python 程序中的测试文件，并对每种输入规模的执行进行计时，计时从提供输入开始，直到解析器产生输出为止。每种输入规模的每次测量由 20 个样本组成。

图 9比较了解析器组合子库和 TyRE 的解析耗时。每张图中，我们绘制了 Idris 2 标准库解析器组合子（橙色）和 TyRE（蓝色）的平均执行时间。我们还为每种颜色添加了较浅的色带，其幅度为该输入规模下抽样运行时间的一个标准差，向两侧延伸。

图 9a：正则表达式： a^n **解析器：** a^n。我们比较了使用正则表达式 n 个连续 a字符的字符串进行解析所需的时间，其中正则表达式为 a^n ，n 从 1 到 1000。两个库的执行时间相近，并且相对于正则表达式和字符串的长度 n 大致呈线性增长。由于每一步都只有一个可能的后续状态，因此 TyRE 在执行过程中始终只维护一个线程。

[8] 我们使用了一颗 Intel Core i7 8700 处理器，主频为 3.2GHz，配备 2666MHz、6 核、65W，搭配 32GB（2×16GB）DDR4 2666 DIMM 内存和 7200RPM SATA-6G ROM。

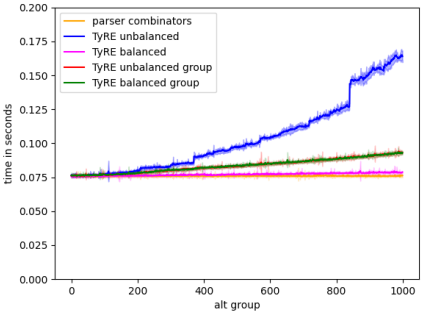


图 10：使用不同的交替模式解析 a 与$a(|a)^{n-1}$。

遍历 Moore 机。同样，解析器组合子解析器逐个消耗字符，无需回溯。

图 9b：正则表达式： $a*$ **待解析字符串：** a^n。这里我们解析同一组字符串 a^n，但统一使用正则表达式 $a*$。同样，两个库相对于 n 均以近似线性的时间运行，且两个库的执行时间相近。

图 9c：正则表达式： $((a*c)|a)*b$ **待解析字符串：** a^nb。在此测试中，TyRE 的扩展性优于 Idris 2 的解析器组合子库。使用解析器组合子构建的解析器在遇到交替时总是首先选择左分支，而在本例中这始终是错误的选择。因此，该示例迫使解析器反复回溯，导致二次复杂度。TyRE 会同时搜索两个分支。得益于通过合并线程来压缩搜索空间，TyRE 只需维护固定数量的线程，从而实现线性复杂度。

为了突出 TyRE 的局限性，下面是 TyRE 的一个病态情形：

图 9d：正则表达式： $a(|a)^{n-1}$ **待解析字符串：** a。TyRE 一次搜索所有可能性并不总是最佳选择。这里，我们用 a 解析单个字符组成的字符串 $a(|a)^{n-1}$。我们用解析器组合子构建的解析器几乎能以恒定时间解析这个字符串。它会选择一个可能的分支，并快速找到与这个单字符相匹配的结果。另一方面，TyRE 会为每条可能的路径创建 n 个线程，并为每条路径各自创建一棵解析树。由于每个选择都用 二选一 类型编码，构建的解析树最多包含 $O(n)$个 二选一 构造器。这种同时遍历方式导致 TyRE 在本测试中的执行时间呈二次增长。

<escape>然而，这个示例具有病态特征，我们通过元编程创建了该正则表达式及其关联类型。图 10 评估了针对这一问题的三种变通方案。我们重复了图 中蓝色部分的评估，参见图9d：一个由正则表达式交替构成的不平衡链。</escape>

我们也可以改为平衡正则表达式中的交替，从而缩短所选分支的编码。例如，对于$n = 8$:

$((a\,|\,a)\,|\,(a\,|\,a))\,|\,((a\,|\,a)\,|\,(a\,|\,a))$，而不是 $((((((a\,|\,a)\,|\,a)\,|\,a)\,|\,a)\,|\,a)\,|\,a$。

这种平衡（洋红色）可将运行时优化为 O($n \log n$)。不过，这种性能提升后的解析器仍比非回溯解析器组合子的对应实现（橙色）慢。

一种更简单的“修复”方法是不再追踪匹配的交替分支。我们对正则表达式应用 `ignore`智能构造器，从而对正则表达式应用`Group` 构造器，得到相应的 TyRE。对不平衡的正则表达式（红色）进行分组，能显著缩短解析时间。这种分组意味着非确定有限自动机将被最小化，所有交替分支都会合并为一条路径。不过，在这种情况下，对平衡的正则表达式（绿色）进行分组反而会降低性能。这种明显的开销来自最小化：在非确定有限自动机遍历的某个阶段，我们会将每个状态与其他所有状态逐一比较，因此又呈现出较平缓的二次增长趋势。`Group` 构造器的真正优势在于自动最小化非确定有限自动机、暂存最小化后的非确定有限自动机并加以复用。

从更定性的角度来看，用户在实际使用中感受到的一大区别是两个库的接口。为了支持正则表达式字面量，TyRE 只解析字符字符串，而解析器组合子库则通用得多。另一方面，使用解析器组合子库编写一个简单的正则表达式匹配器，也需要用户定义更多实体，例如一组词元和一个词法分析器。事实上，Idris 2 的标准库正是因此支持了一份解析器组合子库的副本，专门用于处理字符串。此外，与更为熟悉的正则表达式字面量相比，即使是经过简化、专用于字符字符串的解析器组合子库接口，对新手而言也可能不那么熟悉。

6.2 Sedris

我们在一个流编辑嵌入式领域特定语言中使用了 TyRE — *sedris*。我们以 GNU sed 程序为基础构建 sedris[9] (Pizzini 等人，2020年）。Sedris 脚本是一个按从上到下的顺序执行的命令列表。脚本可以调用“文件脚本”。文件脚本对文件进行操作，并针对每一行依次执行。

我们演示如何使用 sedris 重命名代码中的变量[10]。例如，在 TyRE 的某个假想旧版本中，我们将带类型的“Moore 机”称为“状态机”。我们可以使用 sedris，将所有变量中的子字符串‘`sm`’/‘`SM`’（代表状态机）替换为子字符串‘`mm`’/‘`MM`’（代表 Moore 机）。例如，‘`asSM`’ ↦ ‘`asMM`’ 和 ‘`sm`’ ↦ ‘`mm`’。我们将匹配所有以大写‘`SM`’结尾（`endsWithSM`），或以‘`sm`’或‘`SM`’开头、后面可选跟随一个以大写字母开头的字符串的单词（`startsWithSM`）。为了只匹配完整单词，模式必须以空格作为分隔符。为处理行首和行尾的单词，我们会在匹配前人为地给每行两端加上空格，并在将该行写入文件前去掉这些空格。

使用 TyRE 的 Sedris。图 11 展示了示例 sedris 脚本。完整的 sedris DSL 超出了本手稿的作用域，我们仅解释相关的 sedris 片段。

[9] Lee E. McMahon 于 1973 年左右首次开发了 sed（Hauben 和 Hauben，1997，第 9 章）。[10] 此例仅用于演示。我们不赞成使用简单的基于正则表达式的文本替换来重命名程序变量的绑定和被绑定出现形式。我们坚信，这类操作应交由为此任务精心设计的作用域感知工具完成。只有在没有方便使用的作用域感知工具时，我们才建议使用基于正则表达式的替换。

```
1 rename : Script [<] IO 2 rename =3 -- 用于捕获匹配变量名称的 TyRE。 4 let
tyre : TyRE (Either (Bool, String) String) 5 tyre = letendsWithSM = r " `([a-z]|[
A-Z])+`SM " 6 startsWithSM =r " ((sm)|(SM))!`([A-Z]([a-z]|[A-Z])*)?` " 7 in
startsWithSM <|> endsWithSM 8 -- 逐行执行替换的文件脚本。 9 fileScript : String -
> 文件脚本 [<] 输入输出 10 fileScript fileOut =11 -- 对第一行清空输出文件
12 [ Line 1 ?>ClearFile fileOut 13 -- 在行的两端添加空格 14 , > Exec (\str
=> " " ++ str ++ " ") 15 -- 根据 `toStr' 替换该行中的所有匹配项 16 , > Replace
(AllRe tyre toStr) 17 -- 去掉多余的空格 18 , > Exec ( fastPack . reverse 19 .
dropHead . reverse 20 . dropHead . fastUnpack) 21 -- 将该行写入输出文件 22 , >
WriteTo fileOut]23 in 24 -- 包含所有执行替换的文件的脚本25 [ ["SM.idr"] *
fileScript "MM.idr"26 , ["SMConstruction.idr"] * fileScript "
MMConstruction.idr"27 , ["SMGroup.idr"] * fileScript "MMGroup.idr"
28 ]where 29 dropHead : 列表 字符 -> 列表 字符 30 dropHead [] = []31
dropHead (x :: xs) =xs32 ||| 为匹配项创建替换字符串的函数 33 toStr : 二选一 (
布尔值, 字符串) 字符串 -> 字符串 34 -- 名称开头的 `sm', 例如 smToNFA 35 toStr
(Left (True, str)) = " mm" ++ str ++ " " 36 -- 名称开头的 `SM', 例如
SMConstruction 37 toStr (Left (False, str)) = " MM" ++str ++ " " 38 -- 名称末尾的 `
SM', 例如 asSM 39 toStr (Right str) =" " ++ str ++ "MM "
```

图 11：将 ‘sm’ 替换为 ‘mm’ 的 Sedris 脚本。

sedris 的核心是类型ReplaceCommand 其值告诉 sedris 如何替换子字符串。在这里（第 16 行），我们使用以下变体，它接受一个TyRE 用于匹配，以及一个将每个 match 发送到其替换字符串的替换：

```
AllRe : (re : TyRE a) -> {auto 0 消耗 : IsConsuming re} ->(a-> 字
符串) -> ReplaceCommand
```

为保证生产性，AllRe 还要求证明TyRE 是消耗输入的，即成功的匹配项至少消耗一个输入字符。

Sedris 通过将输入中的每一行连续存储在指定的 模式空间 中来执行这些转换。某些 sedris Command 只有在逐行处理的上下文中才有意义，因此我们根据以下类型为 Command 编制索引：

```
数据ScriptType  = Total | 逐行
```

更复杂的 sedris 命令可以绑定并引用 sedris 变量——这里不作讨论。我们根据命令与上下文中变量的关系为其建立索引：其作用域，类型为 变量 ——一个 SnocList，由 变量构成；以及它们绑定的 列表，由 变量构成。有些命令假设它们与内存中或磁盘上的文件交互，我们使用类型 文件脚本类型 来表示这类假设。因此，sedris 按四个参数为其 Command 类型建立索引：

```
数据 Command : (作用域 :变量) -> (绑定 : 列表 变量) ->ScriptType-> 文件
脚本类型 -> 类型
```

我们将诸如AllRe （第 15 行）的替换命令称为 Replace 构造器：

```
<escape>Replace : ReplaceCommand-> Command sx  [] st io</escape>
```

其索引表示以下保证：

- sx：该命令不对其外围变量作用域作任何假设，因此在任何作用域中都有效；
- []：该命令不会绑定任何新变量；
- st：此 Command 可出现在任何脚本类型中；并且
- io：此 Command 可与任何类型的文件交互。

Sedris 文件脚本还可以调用任意 Idris 字符串转换（第 14 行和第 18 行）：

```
Exec : (字符串-> 字符串) -> Command sx  [] st io
```

我们使用以下 Command（第 22 行）将模式空间写入文件，这要求文件脚本的流支持输出：

```
写入 :  {0 t: 文件脚本类型} -> {auto isIO : 需要 IO t}-> IO 文
件-> Command sx  [] 逐行 t
```

我们可以创建新的输出文件，或删除现有输出文件的内容，以便逐行写入其中（第 10 行）：

```
ClearFile : {0t :文件脚本类型} -> {auto isIO : 需要 IO t}-> IO 文
件-> Command sx  [] 逐行 t
```

将命令组合成文件脚本时，我们可以将其中一些命令限定在特定地址中生效：

- 明确给出的行号；
- 明确的行范围；
- 与正则表达式守卫匹配的行——由一个TyRE给出；或者
- 文件中的最后一行。

我们通过给命令加上地址标签来指定地址：

```
数据带地址命令： (作用域：变量) ->(绑定：列表 变量) -
>文件脚本类型 -> 类型，其中 (>)：Command sx ys 逐
行 t ->带地址命令 sx ys t (?>)：地址 -> Command sx
 [] 逐行 t ->带地址命令 sx  [] t
```

在我们的示例中，我们只在输入文件的第一行（第 12 行）清空输出文件，并使用行地址构造器。其他所有文件脚本命令都作用于每一行。

我们将这些可能带有上下文的命令组合起来，形成适用于文件的脚本，并使用精巧的列表：

```
数据文件脚本： 变量-> 文件脚本类型-> 类型其中空值构造器：文件脚
本sx t(::) ： 带地址命令sx ys t-> 文件脚本(sx  <>< ys) t-> 文件
脚本sx t
```

cons 构造器（::）使用“鱼”运算符——即列表在SnocList上的作用：

```
(<><) : SnocList a -> List a -> SnocList a
```

一个脚本由以下内容构成：ScriptCommand：

```
数据 脚本命令 : (作用域 :变量) -> (绑定 : 列表 变量)-> 文件脚本类型 ->类型
```

例如，我们可以使用这个构造器将文件脚本应用于文件：

```
(*) : 列表 IO 文件 -> 文件脚本 sx 输入输出 -> 脚本命令 sx  [] 输入输出
```

我们使用另一种华丽的列表类型，将顶层脚本命令组合成脚本：

```
数据脚本: 变量-> 文件脚本类型-> 类型where空值构造器: 脚本sx t(::) :
脚本命令sx ys t -> 脚本(sx  <>< ys) t-> 脚本sx t
```

经验报告。编写这个小型 sedris 脚本时，我们使用 TyRE 表达所需的替换没有遇到任何问题。看到计算出的正则表达式类型与预期类型匹配，让我们对该模式充满信心。后来编写 `toStr` 函数时，我们采用了类型驱动的方式，以交互式方法覆盖了所有情况。

TyRE 有两个令人不便之处。首先，TyRE 的标记中缺少用于匹配字符串末尾或开头的特殊标记。这类标记是正则表达式引擎中常见的扩展。在这种情况下，它们本可以让我们不必在每行两侧人为添加空格。其次，模式中所需的方括号数量较多。例如，在 `"((sm)|(SM))!"` 中所有方括号都是必要的，因为在 TyRE 中交替和连接具有相同的优先级，尽管通常连接的绑定更紧。

7 相关工作

我们的工作直接基于 Radanne 的 TyRE 设计（2019）。Radanne 提出了一种带类型正则表达式层，可以添加到现有的正则表达式解析器之上，以确保安全性。他在 OCaml 的 TyRE 库中实现了其设计。我们扩展了他的工作，确保各层始终具备类型安全性，包括我们定制的正则表达式引擎。Radanne 的设计中有一项我们未采用的功能，即解析树反解析。为了支持反解析，我们可以使用 Radanne 原本的 `Conv`转换构造器，该构造器具有双向转换函数。

其他工作也通过为现有正则表达式引擎添加安全层，来确保正则表达式匹配的安全性。Spishak 等人（2012）在 Java 中为正则表达式实现了一种类型系统。该类型系统会静态检查正则表达式的有效性，并确保捕获组结果的数量正确，从而消除“IndexOutOfBoundsException”异常。该实现作为 Checker Framework 的一部分提供，Checker Framework 是 JVM 的一个插件。Blanvillain（2022）提供了类似的安全性保证，同时还将可选部分产生的 `null` 提升为 `Option` 类型。Blanvillain 利用 Scala 3 的匹配类型 （Blanvillain 等人，2022）执行必要的类型层计算。Nair（2021）在 TypeScript 的“typed-regex”库中采用了类似的方法，该库为现有正则表达式引擎提供安全层。该设计使用条件类型（TypeScript 手册，2023）进行类型层计算。该库提供命名捕获组，并保证返回一个包含相关名称的字典。

Weirich（2017 – 2018）在 Haskell 中实现了一个类型安全的正则表达式解析器。虽然她的工作只是一个概念验证，旨在展示依赖类型在 Haskell 中的可能用途，但她的目标与我们的相似。她的实现使用 Brzozowski 导数进行匹配。Cheplyaka（2011–2022）实现了 regex-applicative，这是一个 Haskell 类型安全正则表达式解析器。它使用解析器组合子，因此可以轻松转换结果。目前它还不支持正则表达式字符串字面量。

Willis 等人（2022）实现了 *Oregano*，这是一个 Haskell 中的带类型正则表达式解析器。与我们的实现一样，Oregano 使用带类型 Moore 机和异构栈进行 parse。Oregano 通过生成一个可保证返回正确形状的函数来实现 Moore 机，而不是采用显式的虚拟机实现。一个具体区别是，TyRE 会移除在同一状态相遇的重复线程，从而确保线性性能。相比之下，Oregano 实现了一个回溯解析器，并统计所有可能的歧义结果。Oregano 还使用更先进的分阶段技术来提升性能。

虽然 TyRE 是类型安全的，但其类型并不能保证以下两点：

- 可靠性：只有匹配项才会返回解析树；
- 完备性：所有匹配项都能正确解析。

我们认为，只需稍作努力，就能实现对核心无类型正则表达式的解析可靠性。如果删除“Group”构造器，并将字符谓词的条件改为恰好匹配一个字符，我们认为便能为核心无类型正则表达式实现反解析并证明其可靠性。对于不以性能为重点的经验证正则表达式解析器实现，请参阅 Firsov 和 Uustalu（2013）、Ribeiro 和 Du Bois（2017）以及 Lopes 等人（2016）。

8 结论与后续工作

我们介绍了一个带类型的正则表达式解析器库。借助依赖类型，我们利用正则表达式模式中的信息，在编译时构建额外的安全层，既面向用户，也作用于状态机层面。近年来，针对正则表达式字面量进行静态分析，以验证字面量格式正确并计算其结果类型，已成为一个备受关注的研究主题（Blanvillain，2022；Spishak 等人，2012；Willis 等人，2022；Nair，2021）。安全检查通常是附加在无类型解析器之上的。我们将解析树形状作为所构建解析器的内在特征予以保留。

有趣的是，要提供所有这些保证，并不一定需要依赖类型这类强大的类型系统特性。Haskell 的 Oregano 解析器（Willis 等人，2022）会监控结果类型。要支持具有静态保证的正则表达式字面量，似乎需要类型级计算、大量预处理或宏功能，或是专门的语言支持。最后，在 TyRE 中遍历非确定有限自动机时，如果两个线程处于相同状态，我们就会将它们合并。这种分析使用了带类型非确定有限自动机的深嵌入，采用依赖类型实现尤为直接。这项优化显著提升了性能，并保证所有模式都能在线性时间内解析。

这项工作还有许多可能的扩展。首先，我们可以证明解析器的可靠性，甚至完备性。我们推测，在某些限制条件下，按照我们的架构证明其可靠性会比较直接。这些限制包括不使用用户定义变换、分组以及针对字符的非相等谓词。我们会在解析树中保留证明，以表明字符满足谓词。因此，解析树本身几乎就可以作为可靠性证明。我们只需额外证明解析树确实表示输入词。这一点应当可以自然地从 Moore 机构造中得出。要实现完备性则需要付出更多努力。不仅

我们必须证明：如果 Moore 机中不存在接受路径，那么该词就不匹配该模式；还必须证明，像我们所做的那样删除某些线程，不会影响完备性。

证明 soundness 有一个有用的结果——它免费提供了反解析功能。Radanne 的设计支持反解析。不过，我们可以更直接地实现反解析：为 TyRE 形状变换添加一个双向转换构造器。这样，不包含单向转换的 TyRE 的解析树就支持反解析。

最后，将 TyRE 与 Idris 2 的标准库解析器组合子集成起来，用于词法切分和解析，或者将 TyRE 支持的语言族扩展到 μ-正则表达式（Krishnaswami 和 Yallop，2019）或嵌套词（Alur 和 Madhusudan，2009）。这些扩展将使我们能够引导构建 TyRE，并使用集成的解析器库或 TyRE 本身来解析正则表达式字面量。

致谢。本研究获得英国皇家学会大学研究奖学金和 Enhancement Award 的支持，并获艾伦·图灵研究所种子基金资助。我们感谢 Guillaume X. Allais、Edwin C. Brady、Donovan Crichton、Paul Jackson、James McKinna、Michel Steuwer 和 Jeremy Yallop 提供宝贵建议并展开富有成效的讨论。

参考资料

Alur, R. 和 Madhusudan, P. (2009)，“为字符串添加嵌套结构”，《*ACM*学报》**56**(3), 1–43. **网址：** *https://dl.acm.org/doi/10.1145/1516512.1516518* Asperti, A., Ricciotti, W., Sacerdoti Coen, C. 和 Tassi, E. (2009)，合一中的提示，载于 S. Berghofer, T. Nipkow, C. Urban 和 M. Wenzel 编，《高阶逻辑中的定理证明》，Springer Berlin Heidelberg, 柏林，海德堡，第 84–98 页。Atkey, R. (2018)，定量类型论的语法与语义，载于《计算机科学逻辑研讨会论文集》。Blanvillain, O. (2022)，类型安全的正则表达式，载于《第 13 届 ACM SIGPLAN Scala 研讨会，预印本》。**网址：** *https://github.com/OlivierBlanvillain/regsafe* Blanvillain, O., Brachth¨auser, J. I., Kjaer, M. 和 Odersky, M. (2022)，“使用匹配类型进行类型级编程”，《*ACM*程序设计语言论文集》**6**(POPL)。**网址：** *https://doi.org/10.1145/3498698* Brady, E. (2021)，Idris 2：实践中的定量类型论，载于 A. Møller 和 M. Sridharan 编，《第 35 届欧洲面向对象编程会议（ECOOP 2021）》，第 194 卷，莱布尼茨国际信息学会议论文集（*LIPIcs*），Schloss Dagstuhl – 莱布尼茨信息学中心，德国 Dagstuhl，第 9:1–9:26 页。**网址：** *https://drops.dagstuhl.de/opus/volltexte/2021/14052* Cheplyaka, R. (2011–2022)，regex-applicative，GitHub代码库。**网址：** *https://github.com/UnkindPartition/regex-applicative* Firsov, D. 和 Uustalu, T. (2013)，正则语言的认证解析，载于《计算机科学讲义（含《人工智能讲义》和《生物信息学讲义》子系列）》，第 8307 卷 LNCS。Gundry, A. (2013)，类型推断、Haskell 与依赖类型，博士论文。**网址：** *https://personal.cis.strath.ac.uk/adam.gundry/thesis/thesis-2013-07-24.pdf* Hauben, M. 和 Hauben, R. (1997)，《网民：*Usenet* 与互联网的历史及影响》，Wiley-IEEE计算机学会出版社。

Hopcroft, J. (1971), 有限自动机状态最小化的 n log n 算法，载于 Z. Kohavi 和 A. Paz 编，《机器与计算理论》，Academic Press，第 189–196 页。URL：https://www.sciencedirect.com/science/article/pii/B9780124177505500221 Jiang, T. 和 Ravikumar, B. (1993), “ 最小非确定有限自动机问题是困难的 ”，SIAM 计算杂志 22(6), 1117–1141。URL：https://doi.org/10.1137/0222067 Kameda, T. 和 Weiner, P. (1970), “关于非确定有限自动机的状态最小化”，IEEE 计算机汇刊 C-19(7), 617–627。Krishnaswami, N. R. 和 Yallop, J. (2019), 一种类型化代数解析方法，载于 K. S. McKinley 和 K. Fisher 编，《第 40 届 ACM SIGPLAN 编程语言设计与实现大会论文集，PLDI 2019，美国亚利桑那州凤凰城，2019 年 6 月 22–26 日》，ACM，第 379–393 页。URL：https://doi.org/10.1145/3314221.3314625 Lee, C. Y. (1959年), “用二元决策程序表示开关电路”，贝尔系统技术期刊 38(4), 985–999。Leroy, X., Vouillon, J. 和 Doligez, D. (1996), Objective Caml 系统。Lopes, R., Ribeiro, R. 和 Camar˜ao (2016), 正则表达式的认证基于导数的解析，载于 F. Castor 和 Y. D. Liu 编，《编程语言》，Springer 国际出版，Cham，第 95–109 页。McKinna, J. 和 Wright, J. (2006), epigram 中一个类型正确、栈安全、可证明正确的表达式编译器。McNaughton, R. 和 Yamada, H. (1960年), “自动机的正则表达式与状态图”，IRE 电子计算机汇刊 EC-9(1), 39–47。Mic (2023), TypeScript 手册，条件类型。URL：https://www.typescriptlang.org/docs/handbook/2/conditional-types.html Miller, D. (1992), “混合前缀下的合一”，符号计算杂志。URL：http://www.sciencedirect.com/science/article/pii/074771719290011R Morrisett, G., Walker, D., Crary, K. 和 Glew, N. (1999), “从 System F 到类型化汇编语言”，ACM Trans. Program. Lang. Syst. 21(3), 527–568。URL：https://doi.org/10.1145/319301.319345 Nair, A. (2021), typed-regex，GitHub 代码库。URL：https://github.com/phenax/typed-regex Pizzini, K., Bonzini, P., Meyering, J. 和 Gordon, A. (2020), GNU sed：一种流编辑器，自由软件基金会。Radanne, G. (2019), 无类型正则表达式引擎的类型化解析与反解析，载于《PEPM 2019——2019 年 ACM SIGPLAN 部分求值与程序操作研讨会论文集，与 POPL 2019 联合举办》。Reed, J. (2009), “依赖类型论中的高阶约束简化”，ACM 国际会议论文集系列，第 49–56 页。Ribeiro, R. 和 Du Bois, A. (2017), 认证的位编码正则表达式解析，载于《ACM 国际会议论文集系列》，第 F130805 卷。Sozeau, M. 和 Oury, N. (2008), 一等类型类，载于 O. A. Mohamed、C. Mu˜noz 和 S. Tahar 编，《高阶逻辑中的定理证明》，Springer Berlin Heidelberg，Berlin, Heidelberg，第 278–293 页。Spishak, E., Dietl, W. 和 Ernst, M. D. (2012), 正则表达式的类型系统，载于《第 14 届类 Java 程序形式化技术研讨会论文集》，FTfJP ’12，计算机协会，美国纽约州纽约市，第 20–26 页。URL：https://doi.org/10.1145/2318202.2318207 Thompson, K. (1968年), “编程技术：正则表达式搜索算法”，Commun. ACM 11(6), 419–422。URL：https://doi.org/10.1145/363347.363387 Weirich, S. (2017-2018), 依赖类型正则表达式子匹配。URL：https://github.com/sweirich/dth/tree/master/regexp

Willis, J.、Wu, N. 和 Schrijvers, T.（2022），Oregano：使用 Moore-Cayley 融合对正则表达式进行分阶段处理，载于《第15届 ACM SIGPLAN 国际 Haskell 研讨会论文集》，Haskell 2022，计算机协会，美国纽约州纽约市，第 66–80 页。**网址：***https://doi.org/10.1145/3546189.3549916*