jq 的指称语义与快速解释器

MICHAEL FÄRBER，因斯布鲁克大学，奥地利

jq 是一种广泛使用的工具，提供了一门用于操作 JSON 数据的编程语言。然而，目前 jq 的语义仅由其实现来规定，因此很难对其行为进行推理。为此，我为 jq 语言的一个子集定义了语法和指称语义。特别是，这套语义为解释更新提供了一种新方法。我在一种名为 jaq 的新型 jq 语言解释器中实现了这套语义的扩展版本。虽然 jaq 执行 jq 程序的方式比 jq 简单得多，但在 13 项基准测试中，jaq 有 10 项的速度快于 jq。

arXiv:2302.10576v1 [cs.LO] 21 Feb 2023

1 引言

UNIX 普及了过滤器和管道 [Ritchie 1984]：过滤器是一种从输入流读取数据并写入输出流的程序。管道用于组合过滤器。

JSON（JavaScript 对象表示法）是一种广泛使用的数据序列化格式 [Bray 2017]。JSON 值可以是空值、布尔值、数字、字符串、值的数组，或从字符串映射到值的关联映射。

jq 是一种工具，它提供了一门用于定义过滤器的语言，以及一个用于执行这些过滤器的解释器。UNIX 过滤器处理字符流，而 jq 过滤器处理 JSON 值流。因此，我们可以用相对简洁的过滤器来操作 JSON 数据。例如，以巴黎街道的公开 JSON 数据集 [Paris Data 2022],作为输入，jq 可以使用过滤器“长度”获取街道数量（6508），使用过滤器“`.[].fields.nomvoie`”获取街道名称，并使用过滤器“`[.[].fields.longueur] | 相加`”获取所有街道的总长度（1576813 米）。jq 提供了一门图灵完备的语言，其本身也很有趣；例如，“`[0, 1] | recurse([.[1],相加])[0]`" 可以生成斐波那契数流。因此，jq 成为了一种广泛使用的工具。本文将程序 jq 称为“jq”，将其语言称为“jq 语言”。

jq 语言的语义仅在 jq 手册 [Dolan 2020]中作了非正式说明。然而，文档经常没有涵盖某些情况，或者实现竟然与文档相矛盾。例如，文档称过滤器`limit(n; f)` “从 f 中提取最多 n 个输出”。然而，`limit(0; f)` 会从 f 中提取最多 1 个输出；而当 n为负数时，`limit(n; f)` 会提取 f 的所有输出。尽管这个特定示例很容易修正，但缺乏可供依循的形式化语义这一根本问题依然存在。形式化语义也有助于判断实现中的某些行为究竟是无意造成的，还是有意设计的。

不过，要对 jq 的行为进行形式化规范会非常冗长，因为 jq 有许多优点并不明显的特殊情况。因此，我努力创建了与 jq 的语义十分相似的指称语义（第 3 节），使得在大多数情况下，两者的行为一致，而在较为特殊的情况下则可能有所不同。相较于 jq，一项重要的改进是采用了新的更新语义（第 3.3 小节），这种语义更易于描述和实现，能够消除一系列潜在错误，并支持更高效的执行。

在执行过滤器之前，jq 会先将其编译为一种相对底层的表示形式。基于形式语义，我想了解，一个简单的解释器在处理仅经过少量加工的过滤器抽象语法树时，运行速度能有多快。为此，我实现了一个名为 *jaq* (第 4 节的解释器。要实现高性能执行，一个特别的挑战是如何

作者地址： Michael Färber，因斯布鲁克大学，奥地利因斯布鲁克，michael.faerber@uibk.ac.at。

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

避免数据克隆（第 4.4 小节），以及如何限制克隆的作用范围（第 4.5 小节），从而支持变异。

为了评估 jaq 和 jq，我编写了若干基准测试（第5节）。此外，我还使用一个现有的、以 jq 语言编写的解释器，对 jaq 和 jq 进行了评估，该解释器用于图灵完备语言 Brainfuck。令人惊讶的是，即使采用朴素的执行方式，jaq 在十三项基准测试中的十项上仍比 jq 快。

2 预备知识

本节旨在帮助读者直观了解 jq 的工作方式。在第3节中，我将正式说明与 jq 的语法和语义高度近似的语法和语义。jq 的官方文档见 [Dolan 2020]。

jq 程序称为过滤器。目前，我们先将过滤器视为一个函数：它将一个值映射为一个（惰性的）值流。此外，我们假设值可以是布尔值、整数或值数组。

恒等过滤器“.”返回一个包含输入的流。

jq 支持加法、减法、乘法、除法和取余等算术运算。例如，“. + 1”会返回一个包含输入值后继的流。这里，“1”是一个过滤器，对于任意输入都会返回值 1。

拼接是 jq 中一个重要的运算符：过滤器“f, g”会拼接过滤器 f 和 g 的输出。例如，过滤器“., .”会返回一个包含两个相同输入值的流。

组合是 jq 中最重要的运算符之一：过滤器“f | g”会将过滤器 g 映射到过滤器 f 的所有输出上。例如，“(1, 2, 3) | (. + 1)”会返回 2、3、4。

使用过滤器“[f]”，可以根据 f 生成的流创建数组。例如，过滤器“[1,2, 3]”会拼接过滤器“1”、“2”和“3”的输出，并将其放入一个数组中，得到值 `[1, 2, 3]`。逆过滤器“.[]”在输入是数组时，会返回一个包含该数组中各值的流。例如，对数组 `[1, 2, 3]` 运行“.[]”，会得到由三个值组成的流：1、2、3。我们可以结合这两个过滤器来遍历数组；例如，输入为 `[1, 2, 3]`，时，过滤器“[.[] | (. + 1)]”会返回单个值 `[2, 3, 4]`。由 f 生成的索引所对应的数组值可通过“.[f]”返回。例如，输入为 `[1, 2, 3]`，时，过滤器“.[0, 2, 0]”会返回流 1、3、1。

可以使用过滤器“if f then g else h end”进行分情况处理。对于 f 产生的每个值 v，若 v 为真，该过滤器就返回 g 的输出；否则返回 h 的输出。例如，给定输入 1，过滤器“if (. < 1, . == 1, . >= 1) then . else [] end”返回 `[]`，1，1。

不动点的计算方式如下：给定过滤器 f，“recurse(f)”返回“., (f| recurse(f))”的输出。这样，我们就可以定义一个用于计算阶乘函数的过滤器。

示例 *2.1*（阶乘）。让我们定义一个过滤器 fac，它应当对任意输入数字 n 返回 $n!$。我们将使用过滤器更新的定点来定义 fac。update 的输入和输出应为数组 `[n, acc]`，，并满足不变量：最终输出是 acc 乘以 n 的阶乘。传递给 update 的初始值是数组“[, 1]”。我们可以使用“.[0]”从数组中获取 n，并使用“.[1]”获取 acc。现在可以将 update 定义为“if .[0] > 1 then [.[0] - 1, .[0] * .[1]] else empty end”，其中“empty”是一个返回空流的过滤器。给定输入值 4，过滤器“[, 1] | recurse(update)”返回 `[4, 1]`，`[3, 4]`，`[2, 12]`，`[1, 24]`。不过，我们只对最后一个值中包含的累加器感兴趣。因此可以写成“[, 1] | last(recurse(update)) | .[1]”，其中“last(f)”是一个输出 f 的最后一个输出的过滤器。这样最终会得到单个值 24。

组合也可以用于将值绑定到变量。过滤器“f as $x | g”执行以下操作：给定一个输入值 i，对于将过滤器 f 应用于 i 所产生的每个输出 o，该过滤器都会将变量 $x 绑定到值 o，使其可供 g 使用，并输出将 g 应用于原始输入值 i 所得到的结果。例如，过滤器“(0, 2) as $x | ((1, 2) as $y | ($x + $y))”产生流 1、2、3、4。注意，在这个特定情况下，我们也可以将其写成“(0, 2)+ (1, 2)”，因为诸如“f + g”这样的算术运算符会将 f 和 g 的输出的笛卡尔积作为输入。[1] 不过，在某些情况下，变量是不可或缺的。

示例 2.2（变量是必要的）。jq 定义了一个过滤器“inside(xs)”，它会展开为“. as $x| xs | contains($x)”。这里，我们希望将 xs 作为输入传递给 contains，但同时也希望将传递给 inside 的输入作为参数传递给 contains。没有变量的话，我们就无法同时做到这两点。

对流进行折叠可以使用 reduce 和 foreach：过滤器“reduce xs as $x (init; f)”会保留一个状态，该状态以 init 的输出作为初始值。对于过滤器 xs 产生的每个元素 $x，reduce 都会将当前状态传递给过滤器 f（f 可以引用 $x），然后将状态设为 f 的输出。当 xs 的所有元素都产生后，reduce 返回当前状态。例如，过滤器“reduce .[] as $x (0; . + $x)”计算数组中所有元素的和。类似地，“reduce .[] as $x (0; . + 1)”计算数组的长度。在 jq 中，这两个过滤器分别称为“相加”和“长度”，利用它们可以通过“相加 / 长度”计算数组的平均值。过滤器“foreach xs as $x (init; f)”与 reduce 类似，但它还会产生所有中间状态，而不仅仅是最后一个状态。例如，“foreach .[] as $x (0; .+ $x)”会产生数组所有元素的累积和。

可以使用运算符“|=”来更新值，其功能类似于 Haskell 等语言中的透镜设置器 [Foster 等人，2005 年；Pickering 等人，2017]：直观地说，过滤器“p |= f”会取 p 返回的任意值 v，并将其替换为 f 作用于 v 后的输出。[^haskell-lenses]我们将“|=”左侧的过滤器称为路径表达式。例如，给定输入`[1, 2, 3],`，过滤器“`.[] |= (. + 1)`”会得到`[2, 3, 4],`，过滤器“`.[1] |= (. + 1)`”会得到`[1, 3, 3]`。我们还可以嵌套这些过滤器；例如，给定输入`[[1, 2], [3, 4]],`，过滤器“`(.[] | .[]) |= (. + 1)`”会得到`[[2, 3], [4, 5]]`。不过，并非所有过滤器都是路径表达式；例如，过滤器“1”就不是路径表达式，因为“1”并不指向输入值的任何部分，而是创建一个新值。

诸如“`.[] |= f`”等价于“`[.[] | f]`”，或“. |= f”等价于 f 这样的恒等式，本可以用于定义更新的行为。然而，由于 jq 处理返回多个值的过滤器 f 的方式，这些恒等式在 jq 中并不成立。特别是，当我们向过滤器“. |= (1, 2)”传入 0 时，输出是 1，而不是我们可能预期的 (1, 2)。同样，当我们向过滤器“`.[] |= (., .)`”传入`[1, 2]`时，输出是`[1, 2],`，而不是预期的`[1, 1, 2, 2]`。jq 的这种行为难以定义，也难以推理。这在一定程度上促使我们定义更简单、更优雅的语义：它们在大多数典型用例中与 jq 的行为一致，同时消除上述这类边界情况。

3 规范

本节介绍 jq 语言一个子集的语法和语义。为将形式语法和语义与第 2 节中引入的具体语法区分开来，我们在规范中使用斜体（如“*f*”、“*v*”），而不再使用此前采用的打字机字体（如“f”、“v”）。

[1] Haskell 用户或许会注意到，这两个过滤器与 Haskell 中对应的表达式“`[0, 2] >>= (\x -> [1, 2] >>= (\y -> return (x+y)))`”和“`(+) <$> [0, 2] <*> [1, 2]`”颇为相似，而它们都返回`[1, 2, 3, 4]`。

表 1。按优先级递增顺序给出的二元运算符。用括号括起来的运算符具有相同的优先级。

Name	符号 运算符	
复数	★	“|”、“,”、“⊨”、“或”、“与”
笛卡尔	∘	(=, ≠), (<, ≤, >, ≥), (+, −), (∗, /), %

3.1 语法

一个过滤器 f定义为

$$f \coloneqq n \mid \$x \mid . \mid .[] \mid .[f] \mid [f] \mid (f) \mid f? \mid f \star f \mid f \circ f \mid \text{if } f \text{ then } f \text{ else } f \mid x \mid x(f; \ldots; f)$$

其中 n 是一个整数，而 x 是一个标识符（例如“empty”）。我们将$\$x$ 称为变量。按照惯例，我们用$\$x'$ 表示一个新变量。$\star$ 和$\circ$ 的可能实例见表 1。此外， f可以是形如“f as $\$x \mid f$”的变量绑定，也可以是形如“$\phi\ f$ as $\$x(f; f)$”的折叠，其中 ϕ 要么是“reduce”，要么是“foreach”。

一个过滤器定义的形式为“$f(x_1; \ldots; x_n) \coloneqq g$”。这里， f是一个 n元过滤器，其中 g 可以指代 x_i。例如，我们可以通过定义 true $\coloneqq (0 = 0)$ 和 false $\coloneqq (0 \neq 0)$，来定义生成布尔值的过滤器。

值 v 定义为

$$v \coloneqq \text{true} \mid \text{false} \mid n \mid [v, \ldots, v]$$

其中 n 是一个整数。虽然这只涵盖了 JSON 值的一个子集，但它为语义规范提供了坚实的基础，使其能够相对轻松地扩展到完整的 JSON 值集合。

3.2 语义

制定这些语义的目标按重要性从高到低排列如下：

- 简洁性：语义应易于描述和实现。
- 性能：语义应支持高效执行。
- 兼容性：语义应与 jq 保持一致。

先从几个定义开始。上下文是从变量到值的映射。值结果要么是一个值，要么是一个错误 $\bot$。值结果流记作 $v_0, \ldots, v_n$。两个流 s_1, s_2 的拼接记作 $s_1 + s_2$。

过滤器 f在上下文 c和一个值结果 v下的求值记作 $f|_v^c$，并返回一个值结果流。我们规定，对于任意过滤器 f， $f|_\bot^c = \bot$，并且仅对值定义其求值语义。我们称两个过滤器 f， g等价，当且仅当 $f|_v^c = g|_v^c$对所有 c和 v均成立。

现在我们将介绍几个辅助函数。第一个函数会在流中的所有元素都是值时，将流转换为数组；否则，它会返回流中最左侧的错误[2]：

$$[\langle v_0, \ldots, v_n \rangle] = \begin{cases} [v_0, \ldots, v_n] & \text{if for all } i, v_i \neq \bot \\ v_{\min\{i \mid v_i = \bot\}} & \text{otherwise} \end{cases}$$

[2]在这种简化的语义中，我们只有一种错误，即 $\bot$，因此指定返回哪种错误似乎毫无意义。然而，在实现中，我们可能会遇到不同种类的错误。

表 2. 替换。此处，$\$x\prime$是一个新变量，$\sigma\prime = \sigma\{\$x \mapsto \$x\prime\}$。

φ	$\sigma\varphi$
$., n$（其中 $n \in \mathbb{Z}$），或 $.[]$	φ
$\$x$ 或 x	$\sigma\ (\varphi)$
$.[f]$	$.[\sigma f]$
$f?$	$(\sigma f)\ ?$
$f \bigstar g$	$\sigma f \bigstar\ \sigma g$
$f \circ g$	$\sigma f \circ\ \sigma g$
如果 f那么 g否则 h	如果 σf那么 σg否则 σh
$x(f_1;\ \ldots;\ f_n)$	$x(\sigma f_1;\ \ldots;\ \sigma f_n)$
f 作为 $\$x \mid g$	σf作为 $\$x' \mid \sigma' g$
$\phi\ xs$ 作为 $\$x(init;\ f)$	$\phi\ \sigma xs$作为 $\$x\prime(\sigma init;\ \sigma' f)$

下一个函数有助于定义条件表达式、合取和析取等过滤器：

$$\mathrm{ite}(v, i, t, e) = \begin{cases} \langle\bot\rangle & \text{if } v = \bot \\ t & \text{if } v \neq \bot \text{ and } v = i \\ e & \text{otherwise} \end{cases}$$

最后一个函数用于从列表中获取第 i个元素（如果存在）：

$$v[i] = \begin{cases} v_i & \text{if } v = [v_0, \ldots, v_n] \text{ and } 0 \le i < n \\ \bot & \text{otherwise} \end{cases}$$

为了计算通过定义引入的过滤器调用，我们将替换$\varphi[f_1/x_1, \ldots, f_n/x_n]$定义为 $\sigma\varphi$，其中 $\sigma = \{x_1 \mapsto f_1, \ldots, x_n \mapsto f_n\}$。替换 $\sigma\varphi$定义于表 2：它既应用替换 σ，也会将 φ 中绑定的所有变量替换为新变量。这可以防止 φ 中的变量绑定遮蔽替换 σ值域中出现的变量。

示例 *3.1.* 考虑过滤器“0 as $\$x \mid f(\$x)$”，其中“$f(g)\ :=\ 1$ as $\$x \mid g$”。这里，“$f(\$x)$”会展开为“1 as $\$x\prime \mid \$x$”，其中“$\$x\prime$”是一个新变量。整个过滤器会展开为“0 as $\$x$ |1 as $\$x' \mid \$x$”，其求值结果为 0。如果（错误地）未能将 $f(g)$ 中的 $\$x$ 替换为新变量，那么整个过滤器就会展开为“0 as $\$x \mid 1$ as $\$x \mid \$x$”，其求值结果为 1。

求值语义见表 3。我们假设笛卡尔算子$\circ$ 定义在值对上，并产生一个值结果。我们在第 2 节中看到了所示过滤器的示例。这些语义与 jq 中的实现差异相对较小。一个值得注意的例外是 $f \circ g$，jq 对它的求值方式不同，会将其作为 $\sum_{y \in g|_v^c} \sum_{x \in f|_v^c} \langle x \circ y\rangle$来求值。原因将在第 4.4 小节中说明。请注意，只有当 f和 g 都返回多个值时，才会出现这种差异。

$$\phi_v^c(xs, f) := \begin{cases} \langle\ \rangle + \sum_{x \in f|_v^{c\{\$x \mapsto x\}}} \phi_x^c(xt, f) & \text{if } xs = \langle x\rangle + xt \text{ and } \phi = \text{reduce} \\ \langle v\rangle + \sum_{x \in f|_v^{c\{\$x \mapsto x\}}} \phi_x^c(xt, f) & \text{if } xs = \langle x\rangle + xt \text{ and } \phi = \text{foreach} \\ \langle v\rangle & \text{otherwise} \end{cases}$$

表 3。评估语义。

φ	$\varphi\vert_v^c$
空	$\langle\rangle$
.	$\langle v\rangle$
n （其中 $n \in \mathbb{Z}$）	$\langle n\rangle$
$\$x$	$c(\$x)$
$[f]$	$\left[f\vert_v^c\right]$
f, g	$f\vert_v^c + g\vert_v^c$
$f \mid g$	$\sum_{x \in f\vert_v^c} g\vert_x^c$
f 作为 $\$x \mid g$	$\sum_{x \in f\vert_v^c} g\vert_v^{c\{\$x \mapsto x\}}$
$f \circ g$	$\sum_{x \in f\vert_v^c} \sum_{y \in g\vert_v^c} x \circ y$
$f?$	$\sum_{x \in f\vert_v^c} \begin{cases} \text{如果 } x = \bot \\ \langle x\rangle \quad \text{否则} \end{cases}$
f 且 g	$\sum_{x \in f\vert_v^c} \text{ite}(x, \text{false}, \text{false } g\vert_v^c)$
f 或 g	$\sum_{x \in f\vert_v^c} \text{ite}(x, \text{真}, \text{真 } g\vert_v^c)$
若 f 则 g 否则 h	$\sum_{x \in f\vert_v^c} \text{ite}(x, \text{true } g\vert_v^c, h\vert_v^c)$
.[]	$\begin{cases} \langle v_0, \ldots, v_n\rangle & \text{若 } v = [v_0, \ldots, v_n] \\ \langle\bot\rangle & \text{否则} \end{cases}$
$.[f]$	$\sum_{i \in f\vert_v^c} v[i]$
$\phi\ xs$ 作为 $\$x(init;\ f)$	$\sum_{i \in init\vert_v^c} \phi_i^c(xs\vert_v^c, f)$
$x(f_1;\ \ldots;\ f_n)$	$g[f_1/x_1, \ldots, f_n/x_n]\vert_v^c$ 如果 $x(x_1; \ldots; x_n) := g$
$f \models g$	参见表4

除了表 3中定义的过滤器外，我们还定义两个类似折叠的过滤器“reduce”和“foreach”的语义如下，其中 xs 求值结果为$\langle x_0$、……、 $x_n\rangle$：

$$
\begin{aligned}
&\text{reduce } xs \text{ as } \$x\,(init;\ f) = init \\
&\qquad\qquad \mid x_0 \text{ as } \$x \mid f \\
&\qquad\qquad \mid \ldots \\
&\qquad\qquad \mid x_n \text{ as } \$x \mid f
\end{aligned}
\qquad\qquad
\begin{aligned}
&\text{foreach } xs \text{ as } \$x\,(init;\ f) = init \\
&\qquad\qquad \mid ,\ (x_0 \text{ as } \$x \mid f \\
&\qquad\qquad \mid \ldots \\
&\qquad\qquad \mid ,(x_n \quad \text{作为 } \$x \mid f)\ \ldots)
\end{aligned}
$$

两个过滤器 fold f遍历序列 xs，初始值为 $init$。它们的主要区别在于，“reduce”只返回最终值，而“foreach”还会返回所有中间值。

以下性质可用于消除绑定。

引理 3.2. 设 $\varphi(f)$是一个过滤器，且 $\varphi(f)\vert_v^c$的形式为“$\sum_{x \in f\vert_v^c} \ldots$”。那么 $\varphi(f)$等价于“$f$as $\$x \mid \varphi(\$x)$”。

证明。我们需要证明当 $\varphi(f)$ 分别设为“$f \mid g$”、“$f$as $\$x \mid g$”、“$f \circ g$”、“$f?$”、“$f$and g”、“$forg$”、“if $f$then $g$else h”、“$.[f]$”以及“$\phi\ xs$as $\$x(f; g)$”时，该命题成立。考虑过滤器 $\varphi(f)$

作为 $.[f]$。接下来我们证明 $.[f]$ 等价于 $f\text{as }\$x \mid .[\$x]$:

$$\begin{aligned}(f \text{ as } \$x \mid .[\$x])|_v^c &= \sum_{x\in f|_v^c} .[\$x]|_v^{c\{\$x\mapsto x\}} \\ &= \sum_{x\in f|_v^c}\sum_{i\in \$x|_v^{c\{\$x\mapsto x\}}} \langle v[i]\rangle \\ &= \sum_{x\in f|_v^c}\sum_{i\in\langle x\rangle} \langle v[i]\rangle \\ &= \sum_{x\in f|_v^c} \langle v[x]\rangle \\ &= .[f]|_v^c\end{aligned}$$

$\varphi(f)$ 的其他情况也可以用类似的方法证明。 □

jq 的语义与表 3中所示语义最显著的差异在于更新的情况；也就是说， $f \models g$。我们将在下一小节讨论这一点。

3.3 更新

jq 的更新机制适用于路径。路径是由索引组成的序列 i_j，可以写作$.[i_1]\ldots[i_n]$。它指向一个值，该值可以通过过滤器“$.[i_1] \mid \cdots \mid .[i_n]$”获取。请注意，“.”是一个有效路径，指向输入值。

更新操作“$f \models g$”会先尝试获取 f 返回的所有值的路径，然后对每条路径，将路径上的值替换为 g 作用于该值后的结果。请注意， f不允许生成新值；它只能返回路径。

示例 *3.3*。考虑输入值 $[[1,2],[3,4]]$。我们可以使用过滤器“.[]”从输入中提取数组 $[1,2]$ 和 $[3,4]$，也可以使用过滤器“.[] | .[]”从输入中提取数字 1、2、3、4。要将每个数字替换为其后继，我们运行“(.[] | .[]) $\models$. + 1”，得到 $[[2,3],[4,5]]$。在 jq 内部，首先会构建路径 $.[0][0]$、$.[0][1]$、$.[1][0]$、$.[1][1]$，然后将这些路径上的值更新为 g。

这种更新方法存在几个问题：其中一个问题是，如果 g没有返回输出，收集到的路径可能会指向已经不存在的值。

示例 *3.4*。考虑输入值 $[1,2,2,3]$ 和过滤器“.[] $\models$ g”，其中 g 是“if . =2 then empty else .”，我们可能会认为它会从输入列表中删除所有等于 2 的值。然而，jq 的输出是 $[1,2,3]$。这里发生的情况或许出乎意料，但与上文对 jq 语义的解释一致：jq 会构建路径 $.[0]$、$.[1]$、$.[2]$和 $.[3]$。接下来，它会将 g 应用于所有路径。将 g 应用于 $.[1]$ 会从列表中删除第一个数字 2，留下列表 $[1,2,3]$ 以及待更新的路径 $.[2]$、$.[3]$ 。然而， $.[2]$ 现在指向数字 3，而 $.[3]$ 则指向列表之外。

即使这个特定示例可以通过针对不恰好返回一个输出的过滤器设置特殊情况来正确执行，也存在更一般的示例，而这种方法对它们的处理方式却出人意料。

示例 *3.5.* 考虑输入值 $[[0]]$ 和过滤器“(.[], .[][]) $\models$ g”，其中 g 是“if . =[0] then [1,1] else . + 1”。在 jq 中执行此过滤器时，首先会构建源自“.[]”的路径 $.[0]$，然后构建源自“.[][]”的路径 $.[0][0]$。接下来，在第一条路径上执行 g，得到中间结果$[[1,1]]$。现在，在剩余路径上执行 g，得到的是 $[[2,1]],$，而不是我们原本可能预期的 $[[2,2]]$。

表4。更新语义。这里，$\$x\prime$ 是一个新变量。

μ	$\mu \models \sigma$
空	$\cdot$
$\cdot$	σ
$f \mid g$	$f \models (g \models \sigma)$
f, g	$(f \models \sigma) \mid (g \models \sigma)$
f 作为 $\$x \mid g$	reduce f as $\$x\prime$ (.; $g[\$x\prime/\$x] \models \sigma$)
如果 f 那么 g 否则 h	reduce f 作为 $\$x\prime$ (.; 如果 $\$x\prime$ 那么 $g \models \sigma$ 否则 $h \models \sigma$)
$.[f]$	reduce f as $\$x\prime$ (。; $.[\$x\prime] \models \sigma$)
$.[]$	$[.[] \mid \sigma]$
$x(f_1; \ldots; f_n)$	$g[f_1/x_1, \ldots, f_n/x_n] \models \sigma$ 如果 $x(x_1; \ldots; x_n) \coloneqq g$

这里的一般问题是，过滤器的执行 g 会改变输入值，但只考虑根据初始输入构建的路径。这会导致路径指向错误的数据、指向不存在的数据（这两种情况都出现在示例 3.4中），以及路径缺失（示例 3.5）。

现在我将展示不同的语义，通过交错调用 f 和 g 来避免这个问题。这样一来，这些语义就可以彻底抛弃路径的概念。

这些语义使用一个辅助函数，它接收一个输入数组 v，并将其第 i 个元素替换为对它应用 σ 后得到的输出：

$$(.[i] \models \sigma)|_v^c = \begin{cases} [\langle v_0, \ldots, v_{i-1} \rangle + \sigma|_{v_i}^c + \langle v_{i+1}, \ldots, v_n \rangle] & \text{if } v = [v_0, \ldots, v_n] \text{ and } 0 \le i < n \\ \bot & \text{otherwise} \end{cases}$$

更新语义见表4。 f 作为 $\$x \mid g$ 的情况略微棘手：这里的意图是让 g 能够访问 $\$x$，但 σ 不能。这样做是为了确保与 jq 的原始语义兼容；原始语义会独立执行 μ 和 σ，因此 σ 不应能访问在 μ 中绑定的变量。为确保这一点，我们将 $\$x$ 替换为一个新变量 $\$x\prime$，并在 g 中将 $\$x$ 替换为 $\$x\prime$。

示例*3.6*。 考虑过滤器 0 作为 $\$x \mid (1$ 作为 $\$x \mid .[\$x]) \models \$x$。它会更新输入数组中索引为 1 的元素。如果“⊨”右侧可以访问在右侧绑定的变量，那么数组元素就会被替换为 1，因为变量绑定 0 作为 $\$x$ 会被 1 作为 $\$x$ 遮蔽。然而，由于我们确保右侧无法访问在右侧绑定的变量，数组元素会被替换为 0，也就是最初绑定到 $\$x$ 的值。给定输入数组 $[1, 2, 3]$,，该过滤器会产生最终结果 $[1, 0, 3]$。

总而言之，给定的语义更易于定义和推理，同时在大多数使用场景下保持了与原始语义的兼容性。此外，避免构造路径似乎也能带来更好的性能，正如我将在 第5节 中展示的那样。

4 实现

我为 第3节 中的规范实现了一个解释器。该解释器名为 *jaq*，使用 Rust [Jung 2020] 编写。Rust 是一种函数式系统编程语言，专注于内存安全和性能。

jaq 分为几个部分，即解析器、解释器和命令行界面（CLI）。解释器和解析器不使用 Rust 的标准库，因此可以独立于 CLI 使用，从而能够集成到其他项目中。

内存分配器对执行速度有着显著影响。具体而言，jaq 使用内存分配器 Mimalloc [Leijen 等人 2019] 因为事实证明，与标准内存分配器相比，它能够提升性能。

本节首先介绍 Rust 处理数据共享的方式（第 4.1 小节）。然后，我们将讨论表示过滤器的第4.2小节数据类型，以及表示值的第 4.3 小节数据类型。最后，我们将分析若干过滤器的克隆行为（第 4.4 小节），以及在何种情况下，以严格求值取代惰性求值有助于提升性能（第4.5小节）。

4.1 共享

Rust 和 C/C++ 一样，不会隐式共享数据，也没有垃圾回收器。这意味着默认情况下，复制一个变量会在内存中创建其数据的（深）拷贝。相比之下，在 Haskell 和 OCaml 等许多采用垃圾回收的语言中，将变量赋值给其他变量不会复制这些变量所引用的数据。如果希望在 Rust 中以这种方式共享数据，可以用引用计数指针类型包装数据：例如，Rc<T> 是 C++ 的 shared_ptr<T> 的对应类型，它允许我们共享类型为 T 的数据，并在不再有引用指向该数据时释放它。

列表 1 展示了 Rc 类型 的一个使用示例。具体来说，它展示了 Rc 类型 如何以恒定时间复制值，但代价是只有在独占访问其数据时才能对其进行变异。Rust 禁止我们使用已移动到Rc 类型 中的值（例如 巨大_Vec，位于 列表 1 中）只是该语言确保内存安全的众多安全机制之一。在 C/C++ 等语言中，类似代码可能会在运行时导致未定义行为。

在本节稍后我们将看到，我们既希望共享数据，也可能需要对其进行变异。为了尽可能实现变异，这里的启示是：（1）我们应尽量避免克隆由 Rc 类型 共享的值，从而保留对所含数据的独占访问权。（2）如果必须克隆，我们至少应努力限制克隆的作用域，以便重新获得对数据的独占访问权（如 列表 1 中 b 离开作用域后所示）。我们将在第 4.4 小节讨论如何实现（1），并在第 4.5 小节讨论如何实现（2）。

4.2 过滤器

jaq 将过滤器表示为抽象语法树，其中变量映射到德布鲁因指标[德布鲁因 1972]。这样，我们便可以将上下文实现为一个列表，其 n-th 个元素包含绑定到变量的值 n。上下文是一个由 Rc 共享的链表，能够快速克隆并在表头追加元素。通常， n 非常小，因此查找时间不是问题。

核心过滤器（例如 length 或 sort）由实现语言实现（jq 使用 C，jaq 使用 Rust）。通常，核心过滤器的执行速度明显快于通过定义实现的过滤器。与 jq 不同，jaq 目前不支持定义递归过滤器。因此，jq 中通过递归定义实现的若干过滤器，例如 recurse(f) := ., (f | recurse(f))，在 jaq 中被实现为核心过滤器。另一方面，这使 jaq 能够内联所有对已定义过滤器的调用，从而生成一个没有定义的大型过滤器。

该实现提供了一个用于求值过滤器的函数：它接收一个过滤器 φ、一个上下文 c和一个值 v，并生成一个包含值结果的迭代器 。该函数的类型和实现都与 $\varphi|_v^c$ 第 3.2 小节中展示的求值语义密切对应。过滤器可以在线程之间安全共享。因此，可以使用多个输入并发地对任意过滤器求值。

Rust 核心库中的 Iterator 特征提供了许多实用函数，使语义的实现简洁而高效。例如，Iterator 会通过 size_hint 函数提供其元素数量的下界和（可选的）上界。这些界限会根据评估函数的实现自动推导出来，适用于过滤器输出迭代器。

列表 1。Rc 类型的用法示例。

```
令巨大_向量 = Vec::from([0; 100_000]); // 复
制一个 `Vec` 需要线性时间令副本_向量 = 巨大_
向量.clone();

// 移动 `巨大的_向量` 到一个引用计数指针中……let mut a = Rc::new(huge_
vec);// ……因此之后尝试访问它会引发编译器错误 //let len =huge_vec.len();


// `a`对其数据拥有独占访问权……assert_eq!
(Rc::strong_count(&a), 1);// ……因此我们可以修改 `
a`assert!(Rc::get_mut(&mut a).is_some());

{ // 我们在这里打开一个新作用域 // 克隆一个 `Rc`只需恒定时间，因为它的数据是共享的
let b = Rc::clone(&a); // 然而，现在 `a`失去了对其数据的独占访问权…… assert_eq!
(Rc::strong_count(&a), 2); // ……因此我们既不能通过 `a` 也不能通过 `b`来修改数据
assert!(Rc::get_mut(&mut a).is_none()); } // `b` 即将离开作用域并被销毁…… // ……因
此我们重新获得对 `a`的独占访问权，并可以再次修改它 assert_eq!(Rc::strong_count(&
a), 1);
```

示例 *4.1*。 拼接 f, g 的评估只需一行代码，即可利用现有的 Iterator 组合子链实现。生成的 Iterator 会自动推导其边界；例如， f, g 所产生元素数量的下界，等于 f 和 g 所产生元素数量下界的和。

通常，这些边界并不精确，因为过滤器所产生的元素数量并非总是可以预测的（毕竟过滤器是图灵完备的！）。不过，对于许多简单的过滤器，Iterator 会给出精确的边界。

在将过滤器 f 的输出收集到数组中时，也会用到这些边界，方式是“[f]”；这里会利用 f 所产生输出数量的边界，为输出数组预先分配内存。在 第 4.4 小节中，我们还会看到 `Iterator`边界的另一种用法。

4.3 值

值类型的简化实现见 列表 2。它指出，一个值要么是布尔值、整数，要么是数组。[3] 对于数组，`Rc<T>` 是指向 `T` 类型数据的引用计数指针。对于数组（以及字符串和映射等其他类型），使用 `Rc` 至关重要，因为否则，复制一个 `Val` 所需的时间会与值的大小成正比，而使用 `Rc` 则只需常量时间。

[3]与这种简化的值类型相比，实际的 JSON 值还可以是空值、小数、字符串，或由字符串映射到值的关联映射。

列表 2. 值类型的简化版本。

```
枚举 Val{布尔值(布尔值),
整数(isize),数组(Rc<
Vec<Val>>),}
```

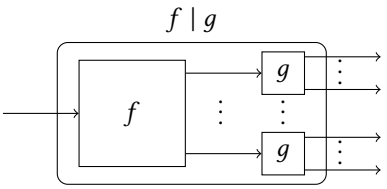


图 1. 过滤器 f 与 g的组合。

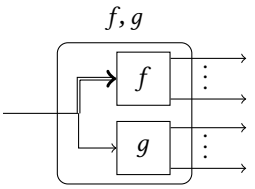


图 2. 过滤器 f 与 g的拼接。

如第 4.1 小节所述，有时我们希望对值进行变异：考虑过滤器“. + [0]”，它会检查输入是否为数组，并返回该数组与 [0]的拼接结果。我们可能会想通过将输入数组与 [0]拼接来直接变异输入数组。然而，这可能行不通，因为输入数组可能有多个引用。一种安全的做法是从输入数组创建一个新数组，使其只有一个引用，然后将其与 [0]拼接。这种做法虽然正确，却非常低效，因为反复与 n 单元素数组拼接，需要创建 n 数组副本，因而会导致二次运行时间。我们可以通过有条件地克隆来解决这个问题：如果数组只有一个引用，就对其进行变异；否则，我们只能先克隆数组，再进行变异。Rust 的 Rc 类型通过 make_mut函数提供了这种模式。这意味着，为了尽可能实现无克隆变异，我们应避免保留值的引用。我们将在第 4.4 小节讨论若干基本过滤器的克隆行为。

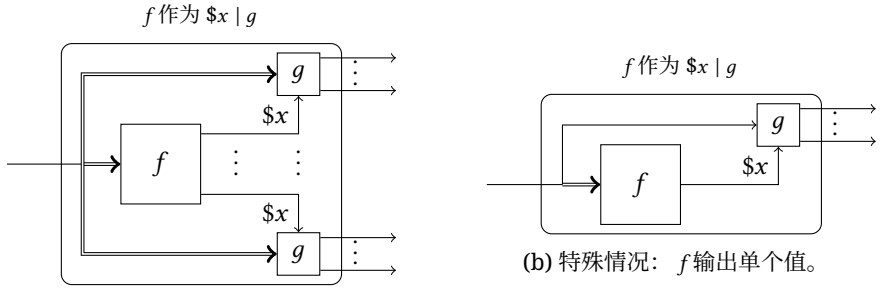
4.4 克隆

本节将考察三个基本过滤器的克隆行为，即组合、拼接和绑定。此外，我们还将讨论一种避免绑定中出现克隆的方法，从而实现 第 4.1 小节中提出的目标 (1)。此处描述的过滤器行为与 第 3.2 小节中的语义一致。

我们用以下方式示意过滤器：一个方框代表一个过滤器，它接收一个值作为输入，并产生任意多个值作为输出。带圆角的方框代表我们即将描述的过滤器。一条带箭头的线表示单个值。双线（例如通向 f的那条线 图 2）表示一个克隆值。

我们先来看 图 1中的组合示意图。这里，输入值被传递给 $f \mid g$，而 f的每个输出都被传递给 f，从而产生 g各个输出的拼接结果，并作为 g 的输出。此过滤器无需克隆，因为每个输入值和输出值最多只会使用一次。

接下来，我们来考察图 2中的拼接。在这里， f、 g的输入会同时传递给 f和 g，而 f、 g的输出则是 f和 g的输出拼接而成的。过滤器 f接收输入值的克隆，而 g 接收原始输入值。这意味着，要在f中修改输入，我们必须克隆它；而在 g中，我们或许可以不经克隆就修改输入（具体取决于是否存在对输入的其他引用）。



(a) 一般情况： f输出多个值。 (b) 特殊情况： f输出单个值。

图 3. 将过滤器 f的输出绑定x 到 g中。

请注意，这一限制源于我们的假设：任何过滤器都可能访问其输入。

示例 *4.2.* 考虑过滤器“(. + [0]), 1”。这里，我们将克隆输入传递给“. + [0]”，因此需要克隆输入数组，以便将 [0] 拼接到其中。此外，我们将原始输入传递给“1”。然而，过滤器“1”显然不依赖其输入，因此原则上，我们可以将原始输入传递给“. + [0]”，使其能够在不克隆的情况下修改输入。

然而，区分使用输入的过滤器和不使用输入的过滤器会大大增加实现的复杂性，因此我没有这样做。

接下来讨论最后一个过滤器，即图3所示的绑定。$x | g 的一般情况为 f，其中我们无法预先知道 f 会产生多少个输出，如图 3a 所示。这里，输入会以克隆的形式传递给 f 和 g 的所有实例，并且 f 的每个输出都会绑定到 $x，再传递给 $g。这意味着我们既无法在 f 中，也无法在 g 中不经克隆地修改输入。如果我们能够预先将 f 的输出数量限定为 n，会怎样？这样，我们就可以将克隆数量从一般情况下的 $n+1$ 减少到 n。这是因为，一旦 f 产生了最后一个值，我们就可以将原始输入传递给 g 的最后一个实例。该实现针对一种特殊情况实现了这一思路：我们知道 f 最多产生 $n = 1$ 个输出，如图3b所示。在这种情况下，我们将克隆输入传递给 f，将原始输入传递给 g。

问题仍然在于，如何预测 f所产生的输出数量。幸运的是，这出乎意料地简单：执行过滤器会产生一个 迭代器，它遍历其输出值；如 第4.2小节所述，迭代器 会提供其元素数量的下界和上界。因此，执行 $f as $x | g 的代码在执行 f时采用一种特殊情况，因为它至少且至多产生一个元素。

4.5 严格性

在本小节中，我们将讨论 jaq 中某些过滤器的严格求值，即加法等笛卡尔运算。这使得 jaq 的求值不同于形式语义（第 3.2 小节），后者并未规定严格求值。作为回报，它缩小了 第 4.1 小节中作为目标（2）所表述的克隆值作用域，从而增加了采用无克隆变异以实现更高性能的可能性。

笛卡尔过滤器 $l \circ r$产生以下值，其中 r求值为 r_1、……、 r_n，而 l求值为 l_1、……、 l_m：

$$\begin{matrix} l_1 \circ r_1, & \ldots, & l_1 \circ r_n, \\ \vdots & \ddots & \vdots \\ l_m \circ r_1, & \ldots, & l_m \circ r_n \end{matrix}$$

假设“。”定义在值上，我们可以用“l as $\$x$ | r as $\$y$ | $\$x$ 。 $\$y$”来朴素地求值“$l$ 。 r”（前提是 $\$x$ 未出现在 r 中）。这种方法有两个缺点：首先，对于 l 产生的每个值 $\$x$，我们都必须重新求值 r，尽管它不依赖于 $\$x$。其次，如图3b所示，$l$ 总是在输入的克隆版本上执行。然而，笛卡尔运算 l 。 r 倾向于只变异 l，因为向数组和字符串的右侧扩展效率更高。这意味着这种方法会损害性能，因为它阻碍了对 l 产生的值进行无克隆变异。也许正因如此，jq 实现了一种类似于“r as $\$y$ | l as $\$x$ | $\$x$ 。 $\$y$”的方法。该方法允许对 l 产生的值进行无克隆变异，但前提是 r 只产生一个输出。此外，这种方法产生输出的顺序不同（也许会令人意外）。jaq 实现了第三种方法：它首先将 r 的所有输出收集到数组 $\$ys$ 中。然后，它才使用原始输入对 l 求值；对于 l 的每个输出 $\$x$，以及 $\$ys$ 中的每个值 $\$y$，它都会产生 $\$x$ 。 $\$y$。与前两种方法不同，这种方法不会重复执行工作，并且无论 r 产生多少个输出，都能对 l 产生的值进行无克隆变异。此外，与 jq 采用的方法不同，这种方法产生输出的顺序与第一种方法相同。不过，它付出的代价是必须严格求值 r；如果 r 产生无限序列的输出，求值就永远不会终止。幸运的是，遇到这种情况时，我们总可以在 jaq 中使用第一种或第二种方法。

5 评估

我通过测量 jq[4] 和 jaq0.8.1 在一组精心构造的过滤器上的运行时间，评估了它们的性能。这些过滤器涵盖了第 2 节中介绍的大部分过滤器。

我在 Ubuntu 22.04 上进行了评估，使用的机器配备四个主频为 2.10 GHz 的 Intel Core i3-5010U CPU 和 8 GB 内存。jq 和 jaq 分别使用 GCC 11.2.0 和 Rust 1.62.0 编译。评估时我只使用了一个核心。

所有基准测试都以一个整数输入 n作为参数。以 n 为参数的基准测试 b 称为 b-n；例如 empty-128。

有两个特殊的基准测试：

- empty：此基准测试由过滤器 empty 组成，该过滤器在没有输入的情况下被调用 n 次。它用于测量启动程序的开销。
- bf-fib：此基准测试使用以 jq 编写的 Brainfuck 解释器运行一个 Brainfuck 程序，该程序用于计算 n 斐波那契数。

其他基准测试都会只调用一次过滤器，并将 n 作为输入值传入。由于生成的值相当大，为避免测量 I/O 速度，过滤器的输出会再经过长度过滤器处理。

- reverse：过滤器“[range(.)] | reverse”会反转数组 $[0, \ldots, n-1]$。这里，“range(n)”返回序列 0, 1, ..., $n-1$。
- sort：过滤器“[range(.) | -.] | sort”会对递减数字数组 $[0, -1, \ldots, -(n-1)]$进行排序。

- add：过滤器“[range(.) | [.]] | add”会构造一个数组的数组，其中每个子数组都只包含一个数字，然后使用 add 将所有单元素数组拼接成一个大型数组。
- kv：过滤器kv定义为“[range(.) | {(tostring): .}] | 相加”，它构造出一个由 n 键值对组成的对象 {"0": 0, "1": 1, ...}。
- kv-update：过滤器“kv | .[] += 1”会递增使用 kv 构造的对象中的值。这里，“p += f”是“p |= . + f”的简写。

[4] jq 源自 https://github.com/stedolan/jq/，修订版 cff5336。此版本的性能显著优于最新的稳定版本 jq 1.6。在 Makefile 中，-DNDEBUG 被添加到 DEFS 中，以省略断言检查。

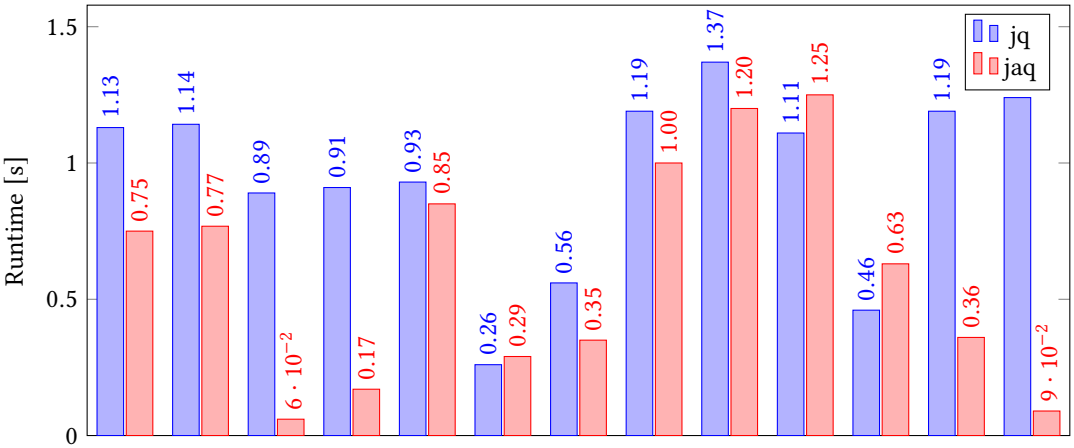


图 4. jq 和 jaq 在一组基准测试中的运行时间（越低越好）。

• kv-entries：过滤器“kv | with_entries(.value += 1)”执行与 kv-update 相同的操作，但它会先将对象解构为由键值对组成的数组，对其进行映射，然后重新构造对象。

• ex-implode：过滤器“[limit(.; repeat("a"))] | add | explode | implode”首先构造一个由 n 个“a”组成的字符串，然后用 explode 将字符串拆解成由其字符组成的数组，接着用 implode 将数组重新构造成字符串。[5]

• reduce：过滤器“reduce range(.) as $x ([]; . + [$x + .[-1]])”会生成一个数组$[x_0, \ldots, x_n]$，其中 $x_0 = 0$ 和 $x_i = i + x_{i-1}$。

• tree-flatten：过滤器“nth(.; 0 | trees) | flatten”会构造一棵深度为 n、叶节点值为 0 的二叉树，然后获取其所有 2^n 叶节点。这里，过滤器 trees 会展开为“recurse([., .])”。

• tree-update：过滤器“nth(.; 0 | trees) | (.. | scalars) |= .+1”会像上一个示例一样构造一棵二叉树，然后将所有叶节点的值加 1。

• to-fromjson：过滤器“"[" + ([range(.) | tojson] | join(",")) + "]" | fromjson”会先构造一个编码数组 $[0, \ldots, n-1]$ 的字符串，然后再将其解析为数组。

结果如图4所示。在十三项基准测试中，有十项 jaq 的速度快于 jq。jaq 在其中一些基准测试中表现更快，部分原因在于 Rust 核心库中的高效数据结构，例如Vec，它在反转和排序基准测试中表现出色。

展示新更新语义威力的一个基准测试是 to-fromjson：在此基准测试中，jq 中耗时最多的操作是 join。在 jq 中，join(sep) 的简化版本[6]定义为“reduce .[] as $x (空值; if . == 空值 then $x else . + sep + $x end)”，

[5]过滤器 explode | implode 是字符串上的恒等映射。[6]jq 实际使用的是 join 的一个更复杂版本，它会根据列表元素的类型将其转换为字符串，不过这对于示例并非必不可少，也不会显著影响性能。

而在 jaq 中，它定义为“`(.[:-1][] += sep) | 相加`”。jq 版本会折叠列表，而 jaq 版本会取出列表中除最后一个元素以外的所有元素（“`.[:-1][]`”），通过加上 `sep` 来更新它们，然后用 `相加` 连接所有列表元素。我们也可以在 jq 中使用 jaq 版本的 `join`；但它的速度比 jq 的版本慢得多。最可能的原因是，jq 会在执行更新前构建完整的索引列表，而 jaq 借助其更新语义（第 3.3 小节）可以省去这一步。

我还对一个用 Go 编写的 jq 实现进行了基准测试，即 gojq 0.12.9[7]。在 tree-flatten 这一项基准测试中，gojq 的速度快于 jaq 和 jq，因为 gojq 的 flatten 过滤器是用 Go 而非 jq 编写的。在 empty 和 to-fromjson-2[16] 两项基准测试中，gojq 的速度也快于 jq。在其他基准测试中，gojq 的速度都慢于 jaq 和 jq。

另一个实现 yq 未纳入评估，因为它的语法与 jq 的语法不兼容。

6 结论

我给出了 jqlanguage 一个子集的形式语法和指称语义，这显著简化了对 jq 过滤器行为的推理。具体而言，我指出了现有 jq 实现处理带路径的更新时存在的不足，并提出了可避免这些问题的新更新语义。我在一个名为 jaq 的解释器中实现了现有语法和新语义；该解释器大量依托 Rust 及其 Iterator 特征构建。为了实现值的无克隆变异，jaq 在可能的情况下会避免克隆，例如在评估变量绑定时。评估结果显示，尽管 jaq 的执行模型显著简单于 jq，但在十三项基准测试中，有十项的速度快于 jq。这表明，我们能够兼得多方面的优势：形式语义、精简的实现以及最先进的性能。

致谢

感谢 Diana Gründlinger 和 Fabian Mitterwallner 对本文草稿提出宝贵意见。本研究部分由奥地利科学基金（FWF） [J 4386]资助。

参考文献

Tim Bray. 2017. JavaScript 对象表示法（JSON）数据交换格式。RFC 8259。https://doi.org/10.17487/RFC8259 Nicolaas Govert de Bruijn. 1972. 使用无名占位符的 λ 演算记法：一种用于自动公式操作的工具，并应用于 Church-Rosser 定理。*Indagationes Mathematicae*（论文集）75, 5（1972），381–392。https://doi.org/10.1016/1385-7258(72)90034-0 Stephen Dolan. 2020. *jq* 手册。https://stedolan.github.io/jq/manual/ 在线；访问日期：2020年12月18日。J. Nathan Foster、Michael B. Greenwald、Jonathan T. Moore、Benjamin C. Pierce 和 Alan Schmitt。2005。双向树变换的组合子：一种针对视图更新问题的语言学方法。载于第*32*届 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集，*POPL 2005*，美国加利福尼亚州长滩，*2005*年*1*月*12*—*14*日，Jens Palsberg 和 Martín Abadi（编）。ACM，233–246。https://doi.org/10.1145/1040305.1040325 Ralf Jung. 2020. 理解并演进 *Rust* 编程语言。博士论文。萨尔大学，德国萨尔布吕肯。https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647 Daan Leijen、Benjamin Zorn 和 Leonardo de Moura。2019。Mimalloc：空闲列表分片实践。载于编程语言与系统——第*17*届亚洲研讨会，*APLAS 2019*，印度尼西亚巴厘岛努沙杜瓦，*2019*年*12*月*1*—*4*日，论文集（计算机科学讲义，第*11893*卷），Anthony Widjaja Lin（编）。Springer，244–265。https://doi.org/10.1007/978- 3-030-34175-6 13_Paris Data. 2022. 现有道路范围名称。https://opendata.paris.fr/explore/dataset/denominations- emprises-voies-actuelles/ 在线；访问日期：2022年9月29日。Matthew Pickering、Jeremy Gibbons 和 Nicolas Wu。2017。Profunctor 光学：模块化数据访问器。艺术、科学与工程编程1, 2（2017），7。https://doi.org/10.22152/programming-journal.org/2017/1/7 Dennis M. Ritchie. 1984. UNIX 系统：UNIX 分时系统的演进。*AT&T* 贝尔实验室技术期刊63, 8（1984），1577–1593。https://doi.org/10.1002/j.1538-7305.1984.tb00054.x

[7]取自 https://github.com/itchyny/gojq/。