arXiv:2303.12254v1 [cs.PL] 22 Mar 2023

P利用部分求值实用地编译 F 表达式 n

F 表达式可在纯函数式 Lisp 中高效取代宏

Nathan Braswell，佐治亚理工学院，美国Sharjeel Khan，
佐治亚理工学院，美国Santosh Pande，佐治亚理工学院，美国

宏是 Lisp 语言中常见的组成部分，也是其中最受赞誉的特性之一。大量研究致力于让宏更安全、更强大，并推动了多个领域的发展，包括保持卫生性和类型化程序分阶段 [Rompf 等人 2013]。然而，宏也有各种缺点，其中之一是它们属于二等公民。对于急切求值的函数式编程而言，尤其严重的是，宏无法作为参数传递给高阶函数，也不能自由组合。

John Shutt 重新定义的 F 表达式 [Shutt 2010], 为宏提供了一种一等且更强大的替代方案，并且与纯函数式编程相得益彰。遗憾的是，直接执行 F 表达式的速度远慢于宏，因为运行时需要重复执行未经优化的操作子组合器代码；而在基于宏的语言中，这些代码会在编译时展开并优化。为了证明 F 表达式可以切实取代宏，我们设计了一种小型纯函数式、基于 F 表达式的 Lisp，*Kraken*，它配备在线部分求值与编译框架，支持一等的部分静态数据环境，并能彻底优化掉那些以宏的风格使用和编写的 F 表达式。我们展示了，该部分求值与编译框架通过消除重复工作并揭示静态信息以实现进一步优化，生成的代码比直接解释快 70,000 倍以上。此外，我们的 *Kraken* 编译器的表现优于现有支持 F 表达式的解释型语言；在一项基准测试中，其 F 表达式性能比 NewLisp 的 [Mueller 2018] 提高了 233 倍。

CCS 概念： • **软件及其工程 → 函数式语言**； **编译器** .

附加关键词与短语：部分求值、Vau、F 表达式、fexpr、WebAssembly

1 引言

Lisp 方言语言 [Flatt and PLT 2010年； Hickey2008； Winston and Horn 1986] 通常具有两种不同的抽象机制：函数和宏。这两种抽象在特性和语义上有所不同。函数在运行时执行，并且总会对其参数求值；宏仅在展开时执行，不会对其参数求值。函数有时可以（取决于实现）在宏中使用，但会受到限制。这些限制将语言一分为二，也使其无法体现函数式编程的若干关键原则——即支持高阶特性和组合。

宏系统通常会通过阻止或设法限制对宏展开的代码环境进行操作，来实现卫生性 [Clinger andRees 1991]。然而，在一些特殊情况下，操作代码环境往往是必要的。通常的解决方案是提供各种逃生口，这会使宏系统变得复杂。出于这些原因，实用的宏系统往往相当复杂，也与其嵌入其中的语言大相径庭。此外，由于宏会在展开后消失，可调试性也会受到影响：程序员必须在脑海中展开宏，才能找出那些并非自己明确编写、而是由宏展开生成的代码中的错误原因。

作者地址：Nathan Braswell，佐治亚理工学院计算机科学学院，美国佐治亚州亚特兰大，
nathan.braswell@gtri.gatech.edu；Sharjeel Khan，佐治亚理工学院计算机科学学院，美国佐治亚州亚特兰大，
smkhan@gatech.edu；Santosh Pande，佐治亚理工学院计算机科学学院，美国佐治亚州亚特兰大，
santosh.pande@cc.gatech.edu。

为解决这些问题，20世纪60年代诞生了 fexpr，它是一种更强大的一等替代方案，后来由 2010 [Shutt 2010]重新加以阐述。按照 Shutt 的术语，fexpr 将函数、宏和内置语言形式统一为一个称为组合器的概念。 *lambda* 引入函数，而 *vau* 引入组合器。组合器能够在调用环境中对其参数求值零次或多次，并且还会接收该环境作为参数，从而能够在调用它的同一作用域中动态求值代码。对参数求值零次的组合器称为操作器，而对参数求值一次或多次的组合器称为应用器。对参数求值一次且不使用调用环境的应用器，就是普通函数。操作器可以涵盖宏、内置特殊形式，甚至还可以涵盖 *if* 和*lambda* 本身这样的控制流形式。

一种基于组合器、使用操作器而非宏的编程语言，具有诸多优势：

• **更简单的语言**：虽然不设独立的宏系统本身就能让语言更简单，但将操作子组合器作为一等值，也进一步简化了语言定义，因为其中没有特殊形式。例如，*if* 和*vau* 只是内置的操作子组合器，由语言定义或假想的解释器循环提供支持。其他通常通过宏实现的语言特性，如*let*、*and*、*or*、*cond*，当然可以实现为派生操作器；进一步说，甚至 *lambda* 也可以派生实现，而非内置。

• **更简单的心智模型与更出色的调试**：将函数和宏（以及过去的特殊形式）统一为一个概念后，调试类宏操作器、在脑海中跟踪其执行过程，都可以像处理普通函数一样。遇到错误时，可以查看堆栈跟踪，并使用调试器检查栈变量，从而找出问题所在。我们在 *Kraken* 中开发的原型调试器，即使相关信息原本会被优化掉，也能通过重新求值用于重建缺失信息的无副作用代码来显示这些信息。在基于宏的语言中，则需要打印宏的不同展开结果，再试着弄清楚究竟哪个展开失败以及失败原因，而且没有调试器支持。

• **更大的灵活性和表达能力**：所有组合器，包括操作器，都是一等对象，因此可以命名、传递给高阶组合器、组合起来，或放入数据结构中。虽然许多语言都有一等函数，但一等宏并未取得同样的成功。组合器兼具两者的特点。例如，在 Scheme 中，*and* 通常是一个会展开为条件控制流的宏，这意味着若不先用 *fold* 包装在 *lambda*中（如清单 1所示），就不能将其传递给诸如 *and* 这样的高阶函数。如果 and 是操作器，则可以直接将其传递给高阶组合器，如清单 2 所示。

```
1  > (fold and #t (列表 #t #t))
2  异常: 无效 语法 and
3  > (fold (lambda (a b) (and a b)) #t (列表 #t #t))
4  #t
```

清单 1. Scheme 的 and 示例

```
1  > (foldl and true (数组 true true))
2  true
```

清单 2. *Kraken*的*and* 示例

fexpr 的强大功能让人想起 Racket 等先进的宏系统 [FlattandPLT 2010], 提倡利用其令人印象深刻的宏系统来定义全新的语言，

不过该宏系统也很复杂。使用 fexpr 的语言可以拥有类似的表达能力，同时语义更简单。清单3 展示了一个 *let1* （一种简单的 let 绑定，仅支持一个变量）和 *lambda* （使用 *let1*）的 fexpr 实现，展现了如何从极为精简的基础语言逐步自举出一门完整语言；在这种基础语言中，*vau* 是唯一的抽象运算符。本文稍后将解释其工作原理。此处仅为展示这样一种思路：许多语言视为原始且内置的特性，也可以在语言内部定义：

```
(( 包装 (vau (let1 )
; lambda 的定义
(let1 lambda (vau se (p b1)
(包装 (eval (数组 vau p b1) se)))
;一个简单的 函数 that 乘以 其参数乘以 two
(lambda (n) (* n 2))
)
; 定义 of let1
)) (vau de (s v b)
(eval (数组 (数组 vau (数组 s) b) (eval v de))
de)))
```

列表 3. *let1* and *lambda* 的 F 表达式实现

尽管有这些优点，基于 fexpr 的纯函数式语言若采用朴素执行方式，速度会极其缓慢。在求值过程中，每次调用组合器时，都会重新执行被调用组合器的函数体；这不仅发生在对应用器进行类似函数的调用时，也会发生在对操作器进行所有类宏调用时。在被调用操作器的定义中，每次组合器调用也都会发生这种重新执行。因此，随着定义链深度增加——也就是类宏操作器在定义中使用其他类宏操作器——重复执行很可能会造成指数级的性能下降。另一方面，宏系统中的宏只会在展开时执行一次，在运行时则不会执行。在一个宏的定义中使用其他宏时，它们都会在编译前完全展开。由于仅凭语法无法判断组合器的参数是会被求值还是以未求值状态传递（因为无法判断该调用针对的是应用器还是操作器），含有 fexpr 的代码很难编译和优化。因此，fexpr 的典型实现不会对其进行优化，而是完全采用解释执行，从而进一步加剧速度缓慢的问题。

像 NewLisp 和 PicoLisp 这样的语言 [Burger 2013年；Mueller 2018]实现了部分 fexpr 的语言通常会限制 fexpr 的嵌套层数，以避免性能逐层下降。他们选择在解释器中直接实现许多组合器，以提升速度，而不是将其编写为派生 fexpr。使用这些语言中的 fexpr 的代码仍会受到性能惩罚，或在达到限制后崩溃。其他研究 [Kearsley [n.d.]；Shutt 2010] 介绍了 fexpr 语言并展示了它们的实用性，但基于这些语言的少数几种实现都极其缓慢。因此，目前还不存在一种主要基于 fexpr、且实用（快速）的语言。

解决 fexpr 性能问题的一种方法是部分求值。人们已为许多语言开发了部分求值器 [Alpuente 等人，1998年；Andersen，1992年；Elphick 等人，2003年；Komorowski，1982年；Lloyd 和 Shepherdson，1991年；Meyer 1991] ，也针对不同的领域 [Berlin，1990年；Berlin 和 Weise，1990年；Futamura 1971]。部分求值的目的是根据部分求值时已知的值对代码进行特化，以便在执行时减少工作量，从而有望提升性能 [Jones，1996年；Würthinger 等人 2017]。部分求值可分为在线技术和离线技术，其中 [Jones 1996] 对二者差异的解释最为清楚。此外，John Shutt [Shutt 2010] 也提出，部分求值或许能解决 fexpr 的

性能问题，但没有提供任何具体细节或实现。尽管在线和离线部分求值技术已广为人知 [Consel 和 Danvy，1993年；Jones 1996]，，目前仍没有适用于 fexpr 的部分求值方案，因此在所有现有实现中，fexpr 依然运行缓慢。

在这项工作中，我们提出了首个完全基于 fexpr 的实用（快速）纯函数式 Lisp 语言：*Kraken*。*Kraken* 采用专为消除所有对类宏操作体的调用而设计的在线部分求值器，并配合一个编译器后端，利用部分求值器揭示的静态信息，生成性能相当不错的 WebAssembly 二进制文件。借助它，我们表明，基于 *fexpr* 的函数式 *Lisp* 可以达到与基于宏的 *Lisp* 大致相当的效率，且表达能力至少不逊于后者。

***Kraken* 语言** (§3)：我们的首项贡献是这门纯函数式、基于 fexpr 的 Lisp 语言，名为 *Kraken*。该语言基于 John Shutt 对纯 Vau 演算的定义 [Shutt 2010]（使用 fexpr，并基于 *vau* 而非*lambda*），同时增添了显式的原始数据和操作，以展示一种更实用的语言。

部分求值与编译器优化 (§4 - §5)：我们的第二项贡献是量身定制的部分求值算法和编译器优化，使基于 fexpr 的函数式语言能够达到有竞争力的性能。部分求值会消除所有行为类似宏的用户定义操作式，而编译器则会内联所有原始操作式（如 if 等），运行时只留下非宏类的应用器。编译器优化（类型推断辅助的原语内联、单次使用闭包内联和惰性环境创建）消除了语言中许多剩余的低效之处。这种结合部分求值与编译的技术表明，类宏操作体经过静态编译消除后，其性能可以与宏相当，这与宏经展开消除的方式类似。

语言评估 (§6)：我们的最后一项贡献是通过若干基准测试评估语言和编译器，以展示其实用性。首先，我们展示了部分求值与编译器优化相结合，可使运行时性能相比基线解释器提升超过**70,000 倍** 。其次，我们展示了 *Kraken*的优化后编译代码，性能显著优于 NewLisp 的解释型 F 表达式 [Mueller 2018] 实现，在一个基准测试中快了 **233 倍** 。最后，我们将运行时性能与 NewLisp 的宏实现进行比较，结果也更快。

论文首先在第 2 节讨论我们的一般编译流程，随后在第 3 节定义 Kraken 语言的简化版本。第 4 节介绍我们的核心贡献：专注于求值消除类宏操作器的部分求值算法。接着，第 5 节讨论与部分求值算法协同发挥作用的主要优化。第 6 节展示我们的基准测试，证明该算法相较于对 fexpr 的朴素解释实现了显著提速；第 7 节列出相关工作，第 8 节则作总结。

2 通用编译框架

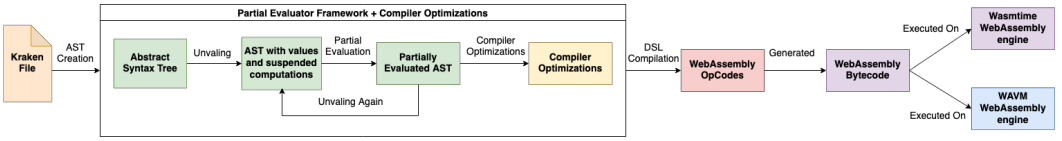


图 1. 编译框架（包括部分求值和编译器优化）将 *Kraken*代码编译为 WebAssembly 二进制文件；对于我们的基准测试，该二进制文件在 WAVM WebAssembly 引擎或 Wasmtime WebAssembly 引擎上执行

Kraken 语言会先编译为 WebAssembly，然后在 Wasmtime WebAssembly 引擎或 WAVM WebAssembly 引擎上执行，如图 1 所示。首先，解析 Kraken 代码以创建抽象语法树（AST）。随后对 AST 进行“标记”，将部分求值所需的簿记信息添加到 AST 中，然后对其进行“unval”处理，标注 AST 中将作为挂起计算进行求值的部分。AST 经 unval 处理后，编译器会检查每个挂起计算，判断其能否基于当前环境（通常包含部分静态数据）求值，从而对 AST 进行部分求值。编译器后端遍历 AST 并生成 WebAssembly 字节码时，会执行基于类型推断的原始组合器内联、单次使用闭包内联、惰性环境创建和基本尾调用消除等优化，进一步提升代码质量和性能。随后，这些字节码可在任何兼容的 WebAssembly 引擎上执行。

3 *KRAKEN*语言

如前所述，所提出的 *Kraken* 语言与 John Shutt 对纯 Vau 演算的定义十分接近（[Shutt 2010]），并在此基础上增添了显式的原始数据类型和操作，从而构成一种更实用的语言。我们以纯 Vau 演算为基础，而非以构成 John Shutt 的 Kernel 语言基础的复杂得多的演算为基础，这使部分求值和优化更简单，也更可行。其复杂演算中主要缺少的特性是可变性和续延。为了使我们的形式化易于理解，我们展示了一种比实际语言进一步简化的演算，其中去除了布尔值、变长参数，并精简了内置原始组合器。

我们首先介绍 *Kraken* 语言的语法、上下文和小步语义，然后阐述其表层语法与内部语法及语义之间的区别。

3.1 语法

$$n \in \mathbb{N} \text{ (整数)} s \in Symbols_0 \in \{\text{1 eval,0 vau,1 wrap,1 unwrap, 0 if0,0 vif0, 1 int-to-symbol,1 symbol?,1 int?,1 combiner?,1 env?, 1 array?, 1 len,1 idx, 1 concat, 1 +,1 <=}\} \text{ (原始操作)} E := (s \leftarrow T) \ldots \mid, E \mid (s \leftarrow T) \ldots \mid s' \leftarrow E, E \text{ (环境)} A := (T \ldots) \text{ (数组)} C := \text{comb } n\ s'\ E\ (s \ldots)\ T \text{ (组合器)} S := n \mid o \mid E \mid C \text{ (自求值项)} V := S \mid s \mid A \text{ (值)} T := V \mid AT \text{ (项)} AT := [\text{eval } T\ E] \mid [\text{combine } T\ (T \ldots)\ E] \text{ (活动项)}$$

图 2. 基础语言的语法

该语言的表层语法（图 2）由符号和整数构成的数组组成。内部语法包含原始组合器和派生组合器、环境以及主动执行项。正是表层语法与内部语法之间的这种划分，使得建立非平凡的理论成为可能 [Shutt2010]。

3.2 上下文

$$\mathrm{E} := \quad \square \mid [\text{combine } \mathrm{E}\ (T \ldots)\ E] \mid [\text{combine } T (\mathrm{E}\ T \ldots)\ E] \mid [\text{combine } T (T \ldots, \mathrm{E}\ T \ldots)\ E] \mid [\text{combine } T (T \ldots, \mathrm{E})\ E]$$

图 3. 基础语言的上下文

基础演算中的求值上下文（图 3）定义了当前必须进行求值的空位，即求值数组的头部以找到要调用的组合器，或在调用组合器之前对参数求值。

3.3 小步语义

$$\mathrm{E}[E] \rightarrow \mathrm{E}[E'] \text{ (if } E \rightarrow E') \quad [\text{eval } S\ E] \rightarrow S[\text{eval } s\ E] \rightarrow \mathit{lookup}(s, E) \quad [\text{eval } (T_1\ T_2 \ldots)\ E] \rightarrow [\text{combine } [\text{eval } T_1\ E]\ (T_2 \ldots)\ E] \quad [\text{combine comb } (S\ n)\ s'\ E'\ (s \ldots)\ Tb\ (V \ldots)\ E] \rightarrow [\text{combine comb } n\ s'\ E'\ s\ Tb [\text{eval } V\ E] \ldots\ E] \quad [\text{combine comb } 0\ s'\ E'\ (s \ldots)\ Tb\ (V \ldots)\ E] \rightarrow [\text{eval } Tb\ (s \leftarrow V) \ldots |s' \leftarrow E,\ E'] \quad [\text{combine } (S\ n)\ \mathrm{o}\ (V \ldots)\ E] \rightarrow [\text{combine } n\ \mathrm{o}\ ([\text{eval } V\ E] \ldots)\ E]$$

图 4. 基础语言的语义

我们的小步语义（图 4）提供了调用、符号和值的求值步骤。*Kraken* 会将自求值值求值为其自身，并在当前环境中查找符号，以对符号求值。最有意思的情况是调用，因为它要求先对数组的第一个元素求值，从而得到一个组合器。

获得要调用的组合器后，调用如何继续取决于“包装层级”和组合器的类型。“包装层级”指示组合器的参数应被求值多少次，而语义则编码了对参数进行相应次数的求值，并确保组合器在调用前的“包装层级”为 0（每次对参数求值时都会递减）。正如我们之前提到的，包装层级为 1 的组合器是应用器，相当于其他语言中的函数。组合器可以是原始组合器（内置于语言中），也可以是派生组合器（用户通过调用 *vau* 创建）。派生组合器的函数体在一个新环境中求值，该环境将参数符号映射到实际参数，包括将一个特殊符号映射到当前动态调用环境，并链接到派生组合器中保存的现有静态环境。下一节将进一步说明原始组合子。

3.4 小步语义（选定原语）

最重要的原始组合器的语义见图5，为便于理解，下面还用英语作了进一步说明：

- 0 **eval**：在给定的环境中对其参数求值。

[combine 0 eval (V E') E] → [eval V E'][combine 0 vau (s' (s ...) V) E] → comb 0 s' E (s ...) V[combine 0 wrap comb 0 s' E' (s ...) V E] → comb 1 s' E' (s ...) V[combine 1 unwrap comb 1 s' E' (s ...) V E] → comb 0 s' E' (s ...) V[combine 0 if0 (V_c V_t V_e) E] → [combine 0 vif0 ([eval V_c E] V_t V_e) E][combine 0 vif0 (0 V_t V_e) E] → [eval V_t E][combine 0 vif0 (n V_t V_e) E] → [eval V_e E] (n != 0)[combine 0 int-to-symbol (n) E] → 'sn（由数字 n 构成的符号）[combine 0 array (V ...) E] → (V ...)

图 5. 基础语言原语的语义

- 0 **vau**：创建一个新的组合器，类似于其他语言中的 lambda，但其“包装层级”为 0，这意味着所创建的组合器不会对其参数求值。
- 0 **wrap**：递增其参数的包装层级。具体来说，我们将一个包装层级为 n 的组合器（可能是由*vau*创建的包装层级为 0 的组合器）“包装”起来，从而创建一个包装层级为 n+1 的组合器。包装层级为 1 的组合器类似于其他语言中的常规函数。
- 0 **unwrap**：递减传入组合器的“包装层级”，是*wrap*的逆操作。
- 0 **if**：仅对其条件求值，并在下一步转换为0 **vif**原语。由于存在不终止的风险，它无法对两个分支都求值。

- 0 **vif**：根据条件是否非零，对两个分支之一求值并返回该分支。
- 0 **整数转符号**：根据一个整数创建符号。
- 0 **数组**：返回由其参数列表构成的数组。

下面只介绍一些不那么有趣的原语：

- [combine 0 **类型测试?** (A) E]：数组？、组合？、整数？和符号？，每个原语在唯一参数属于对应类型时返回 0，否则返回 1。
- [combine 0 **长度** (A) E]：返回唯一数组参数的长度。
- [combine 0 **idx** (A n) E]：返回数组 A 中第 n 个元素。
- [combine 0 **concat** (A B) E]：将两个数组参数合并为一个串接后的数组。

- [combine 0 + (A A) E]：将其参数相加
- [combine 0 <= (A A) E]：若其参数按递增顺序排列，则返回 0；否则返回 1。

3.5 基础语言概要

上文定义的这种基础演算不仅能够借助原语和派生的用户应用器执行标准的 lambda 演算运算，还能通过支持操作器来支持一系列超出宏式行为的功能。引言中列出的所有优点都适用于这种演算，性能方面的缺点也同样如此，至少在朴素实现的情况下是这样。我们的部分求值与编译框架将展示如何把这种基础语言编译为性能尚可的二进制文件（在我们的原型中为 WebAssembly 字节码）。

4 部分求值

部分求值至少可以追溯到 Lombardi 为 Lisp 开发的部分求值器 [Lombardi 1964]。在这项工作中，我们设计了一种部分求值器，专门用于部分求值并消除行为类似宏的 fexpr，我们称之为 **类宏操作器**。通过对这些 fexpr 进行部分求值和内联

调用，以类似于宏展开的方式，我们证明了基于 fexpr 而非宏的语言可以达到近似相同的效率，且至少同样强大。我们的部分求值算法可以消除对所有类宏操作器的使用；所谓类宏，是指其使用方式是静态的，且组合器定义大致类似于条目 4。

```
1 (let (helper (lambda (...)  <生成类似宏的代码 >))2 (vau dynamic_env (parameters) (eval (helper
pa
```

在这些情况下，组合器接收未经求值的参数（因为它是一个操作器），将它们传递给一个辅助函数，该函数根据传入的代码参数生成新代码，然后立即求值由类宏函数生成的代码。按照此模板编写的任何组合器都会被我们的算法部分求值消除。我们的部分求值器和编译器能够处理的情况不止这一特定的“类宏”模板，但我们重点讨论这一情况。通过证明，任何以宏形式编写的内容（会被展开消除）都可以移植为与之对应的操作子组合器（会被部分求值消除），我们确立了使用 fexpr 而非宏作为纯函数式类 Lisp 语言基本构建模块的实用性。创建新语言时，选择 fexpr 作为函数与宏组合的基础构造，既不会损失表达能力，也不会造成显著的性能损失。任何未被部分求值消除的 fexpr，都是无法通过宏表达的内容，因此在这些情况下付出性能损失并不严重。

我们为 *Kraken* 实现了一种在线部分求值策略。基于 [Jones 1996] 的描述，离线部分求值要求事先完成绑定时间分析，以便确定系统中哪些代码片段依赖于动态运行时值，哪些可以静态计算。另一方面，在线部分求值会实际执行程序：若实值已知为静态值，就使用实值；否则就使用符号值。只使用静态值的计算会产生更多已知的静态值，而涉及动态值的计算则会生成残余代码，供最终程序使用。

在我们的情况下，如果不至少执行一部分程序，就连将使用哪些符号作为变量都无法确定，更不用说判断这些变量包含的是静态值还是动态值了，因此在线部分求值是我们的解决方案。称其为在线部分求值算法，并不是说它在程序运行时执行；实际上，它是在编译时运行的。换句话说，“在线”仅指绑定分析相对于部分求值何时进行，而不是指部分求值本身何时进行。

复杂的部分求值算法能够处理部分静态数据结构，其中只有部分数据和结构是已知的 [Sperber 和 Thiemann 1996]。部分静态数据的经典示例是 Lisp 序对，其中一侧包含一个静态整数，另一侧则是残余代码。一些高级部分求值器会维护有关值和残余代码的额外信息，例如它们的类型 [Ruf 1993]。

以下是消除类宏操作器时的关键难点：必须将调用点环境具体化并传递给操作器，而具体化后的环境将是一个部分静态数据结构。也就是说，环境中的每一帧要么是静态数据，将符号映射到静态值；要么是对动态数据的静态描述，其中包含符号以及创建该数据的组合器的标识符（通过调用创建）。这两种类型的帧都可以继续向上链接到父帧，而父帧也可能属于任一类型。如何正确处理这些部分静态数据环境，同时确保所有类宏操作器都被部分求值消除，并避免不终止和指数级运行时间，是这项工作的关键挑战。一个棘手的问题是，组合器定义可能需要在静态数据量不同的环境中被部分求值多次，直到它拥有

足够的信息，能够按照我们的标准尽可能地化简，也就是消除所有静态已知的操作器调用。这意味着，如果某个形式的部分求值失败，我们不能立即将其转换为残余代码，因为之后它可能会再次被求值。如果采用简单的做法，很容易导致指数级运行时间：每次调用时，先对组合器求值，再对参数求值，随后又对组合器求值。由于这种开销会在组合器内部的每次调用中不断累积（还会通过包含组合器的环境、以及包含环境的组合器层层累积，且没有记忆化或类似机制），因此需要一种缓解手段。我们解决这一问题的方法是进展所需 ID 系统，下面将对此进行说明。

随着部分求值的各个方面，尤其是与调用相关的部分，变得日益复杂，我们将算法拆分成多个章节，以便更容易理解。我们会在每个章节中穿插示例，并在最后给出一些最终示例来展示整个过程。部分求值路线图如下：

- **第 4.1 节**：部分求值上下文、标记语法和 Unval 关系
- **第 4.2 节**：小步语义
- **第 4.3 节**：辅助关系
- **第 4.4 节**：原语的部分求值版本
- **第 4.5 节**：部分求值的总体效果
- **第 4.6 节**：示例

4.1 求值上下文、标记语法和 Unval 关系

对于部分求值，我们添加了一个新的活动项 *under*，将 *eval* 替换为 *peval*（部分求值），并向 *combine*中添加新值，如图6所示。

AT:= [peval $T/i_x/E\ ES\ FS$] [under $T(T\ldots)\ /i_x/$ $E\ ES\ FS$] [combine $T(T\ldots)\ E$] (活动项)

图 6. 部分求值的新项

在整个部分求值过程中，我们会跟踪两类关键信息：包含推进求值所需值的环境的标识符，以及当前正在求值的项的集合。这使我们能够大幅裁剪执行空间，并引导部分求值的路径。

这些簿记信息通过一个“标记”遍历添加。部分求值会基于这个经过“标记”的表示进行，编译器也一样。簿记为每种形式添加的额外信息如下：

- **组合器**：一个唯一标识符（i），用于标识调用此组合器所创建的环境。示例：$/i/$ **comb** $n\ s'\ /i'\ f_\prime/E'\ (s\ldots)\ Tb$
- **环境**：一个唯一标识符（i_r 或 i_f），用于将此环境与创建它的调用所对应的组合器关联起来。下标 r 或 f 表示该环境是“虚拟”的还是“真实”的：“虚拟”环境是对动态环境的静态描述，它将符号映射到占位值；“真实”环境则是从符号到（几乎全是）静态值的完全静态映射。之所以说“几乎全是”，是因为“真实”环境可以将符号映射到“虚拟”环境，例如部分求值一个接收其调用点环境的操作器调用时就会出现这种情况。示例：$/i_r/(s \leftarrow V)\ \ldots\ |s' \leftarrow /i_x/E,\ /i'_{x\prime}/E'$ 示例：$/i_f/(s \leftarrow V)\ \ldots\ |s' \leftarrow /i_x/E,\ /i'_{x\prime}/E'$
- **符号**：符号将在其中解析的环境标识符（如果适用）。True 表示该符号尚未部分求值，因此环境标识符将解析为

未知。 ∅ 表示该符号是一个值，而不是挂起查找。示例：/true/*s*示例：/7/*s*示例：/∅/*s*

- **数组**：数组可以通过以下三种选项之一进行标记：值、*freshCall* 或 已尝试调用。值表示该数组是一个值。*freshCall* 表示该数组是一个尚未经过部分求值的调用。已尝试调用表示该挂起调用已经过部分求值，但无法继续。已尝试调用 包含两个值 —— 如果被调用的组合器需要动态调用环境，则为其标识符（否则为 ∅），以及/或者捕获并阻止无限递归的（形式、环境）对。示例：/值/(V_1 V_2) 示例：/freshCall/(V_1 V_2) 示例：/atmdCall ∅ ∅/(V_1 V_2)

在我们的实现中，为提高效率，我们会存储额外的数据，例如在数组内部找到的所有进度标识符；但为了便于说明，我们会在附录 A 中将它们定义为独立函数。我们的实现本质上会预先计算这些函数的结果，并将其作为额外的书签数据存储起来，从而将指数级查找开销降为常数级。这也可以通过记忆化实现。

图7 中的标记关系只需在表层语法中添加记账信息，因此相当简单。

$$\begin{aligned}
\text{mark}(n) &= n \\
\text{mark}(s) &= /\emptyset/s \\
\text{mark}((T\ldots)) &= /\text{val}/(\text{mark}(T)\ldots)
\end{aligned}$$

图 7. 标记关系

我们还必须将求值拆分为两个部分：“取消求值”和“部分求值”。取消求值将一个值转换为挂起计算。部分求值则对挂起计算（或包含挂起计算的值）进行处理，尝试至少执行一部分求值，以减少运行时的工作量。现在，求值就是先取消求值，再进行部分求值。

$$\begin{aligned}
\text{unval}(n) &= n \\
\text{unval}(\langle n\ \mathbf{o}\rangle) &= \langle n\ \mathbf{o}\rangle \\
\text{unval}(/i''/\langle \mathbf{comb}\ n\ s'\ /i'_{f'}/E'\ (s\ldots)\ Tb\rangle) &= /i''/\langle \mathbf{comb}\ n\ s'\ /i'_{f'}/E'\ (s\ldots)\ Tb\rangle \\
\text{unval}(/i/E) &= /i/E \\
\text{unval}(/\emptyset/s) &= /\text{true}/s \\
\text{unval}(/\text{val}/(T_1\ T_2\ldots\ T_n)) &= /\text{freshCall}/(\text{unval}(T_1)\ T_2\ldots\ T_n)
\end{aligned}$$

图 8. Unval 关系

取消求值（图8）一个自求值值时，结果就是该值本身，因为没有需要挂起的计算。取消求值时会发生变化的两种形式是符号（它们会从值变为对环境中该符号的挂起查找）和数组（它们会变为挂起调用）。

图 9 中的上下文定义方式与之前相同，但新增了“peval”（部分求值）、组合器体，以及新“under”形式中的多个位置。

$$\mathrm{E} := \quad \square \mid [\text{peval } \mathrm{E}\ /i_x/E\ ES\ FS] \mid \text{comb } n\ s'\ /i_x/E\ (s \ldots)\ \mathrm{E} \mid [\text{under } \mathrm{E}\ (T \ldots)\ /i_x/E\ ES\ FS] \mid [\text{under } T(\mathrm{E}\ T \ldots)\ /i_x/E\ ES\ FS] \mid [\text{under } T(T\mathrm{E}\ T \ldots)\ /i_x/E\ ES\ FS] \mid [\text{under } T(T \ldots \mathrm{E})\ /i_x/E\ ES\ FS] \mid [\text{combine } \mathrm{E}\ (T \ldots)\ E] \mid [\text{combine } T(\mathrm{E}\ T \ldots)\ E] \mid [\text{combine } T(T \ldots, \mathrm{E}\ T \ldots)\ E] \mid [\text{combine } T(T \ldots, \mathrm{E})\ E]$$

图 9. 部分求值的上下文

4.2 部分求值小步语义

在附录 B 中，我们提供了一些简化的伪代码，作为本文所述关系的补充参考和路线图。

为简便起见，我们将语义分为两个小节。第一小节讨论除调用之外所有内容的部分求值。第二小节则专门讨论调用，因为它们较为复杂。

4.2.1 非调用的部分求值。：非调用的部分求值更容易理解，也有助于熟悉部分求值的基本语义。

$$\begin{array}{rcl}
\mathcal{E}[E] & \to & \mathcal{E}[E'] \text{ (if } E \to E') \\
\\
[\text{peval } x\ /i_x/E\ ES\ FS] & \to & x \text{ if true } \notin \text{neededIDs}(x) \wedge \\
 & & \text{neededIDs}(x) \cap ES = \emptyset \wedge \\
 & & \text{resumeForms}(x) \cap ES = \text{resumeForms}(x) \\
 & & \textbf{else:} \text{ continue below} \\
\\
[\text{peval } /x/s\ /i_x/E\ ES\ FS] & \to & lookup(s, E) \\
[\text{peval } /i'_{x'}/E'\ /i_x/E\ ES\ FS] & \to & /i'_{x'}/E'' \textbf{ if } /i'_{x''}/E'' \in ES \textbf{ else } /i'_{x'}/E' \\
\\
[\text{peval} & & \\
/i''/\langle \textbf{comb } n\ s'\ /i'_r/E'\ (s \ldots)\ Tb \rangle & & \\
/i_x/E\ ES\ FS] & \to & /i''/\langle \textbf{comb } n\ s'\ /i'_r/E'\ (s \ldots)\ Tb \rangle \\
\\
[\text{peval} & & \\
/i''/\langle \textbf{comb } n\ s'\ /i'_{f'}/E'\ (s \ldots)\ Tb \rangle & & \\
/i_x/E\ ES\ FS] & \to & \text{let } E'' = \langle \langle (s \leftarrow /i''/s) \ldots \mid s' \leftarrow /i''/s',\ /i_x/E \rangle \rangle \text{ in} \\
 & & /i''/\langle \textbf{comb } n\ s'\ /i_x/E\ (s \ldots) \\
 & & \quad [\text{peval } Tb\ /i''_f/E''\ \{/i''_f/E''\} \cup ES\ FS] \rangle
\end{array}$$

图 10. 非调用部分求值语义

在图 10中，我们可以看到除调用以外所有形式的部分求值形式关系。正在主动进行部分求值的形式 $[\text{peval } x\ /i_x/E\ ES\ FS]$ 的组成部分如下：

- 需要进行部分求值的形式（图中的 x ）。
- 对其进行部分求值的环境（图中的 $/i_x/E$ ）
- 用于对当前形式上方的形式求值的环境“调用栈”（此处为集合；见图中的 ES ）

用于检查并防止无限递归的当前正在求值的形式集合（见图中的 FS）

我们的第一个关系与基础语言的小步语义相同——如果一个较大表达式中的某个子组件（由 Context 形式指示）可以进行一步求值，那么这个较大表达式也可以进行一步求值。

下一个关系用于检查对该形式进行部分求值是否会取得进展。以下三个条件若有一个成立，就意味着该形式无法取得进展：

- 该形式已经部分求值（$\text{true} \notin \text{neededIDs}(x)$）
- 该形式取得进展所需的环境中，没有任何一个当前是真实的或位于我们的环境栈中（$\text{neededIDs}(x) \cap ES = \emptyset$）
- 该形式此前并未为了避免无限递归而停止求值，或者我们当前已不在启动该循环的形式之下（$\text{resumeForms}(x) \cap ES = \text{resumeForms}(x)$）

如果这三项都成立，那么对该形式进行部分求值不会产生任何影响，因此可以立即返回该形式。

另一方面，如果其中任何一项不成立，该形式就可能可以继续推进，因此会继续检查剩余的关系。请注意，整数、符号值和数组值都已经由进度检查条件返回。剩下的只有挂起的符号查找、环境值或派生组合器。对于挂起的符号查找，我们只需在当前环境中查找该符号。如果此环境是伪环境，就会返回同一个符号，并标记上它在真实环境中对应环境的标识符。对环境值进行部分求值时，我们会检查当前环境_栈，查看其中是否有更新的真实版本（由标识符标识）；如果有，就返回该版本，否则原样返回该环境。

派生组合器是本节中最复杂的情况。首先，我们检查它的静态环境是否为真实环境。如果是，根据图 10 的第五条规则（静态环境中的下标 r 表示这一点 $/i'_r/E'$，即组合器值中的静态环境），我们会立即返回它，因为该组合器已经在真实环境中求值过。作为一个闭包，它在创建时捕获了原始求值所用的环境；由于该环境之后可能已经移动，因此不应替换它。另一方面，伪静态环境（静态环境中的下标 f 表示这一点 $/i'_f/E'$，即组合器值中的静态环境）意味着闭包尚未进行初始求值，并且仍处于最初定义的位置。在这种情况下，我们用当前环境替换旧的静态环境，然后创建一个新的伪环境，将函数的参数映射到标记有组合器/伪环境标识符的占位符号。接着，我们在该伪环境中对组合器的函数体进行部分求值。最终结果是一个新的派生组合器形式，其中封装了新的静态环境和部分求值后的函数体形式。

4.2.2 调用的部分求值。 由于需要维护大量簿记信息，调用的部分求值更加复杂。这种复杂性可见于图 11。

首先，我们对数组中的第一个元素进行部分求值，它将是组合器；接着转换为 *combine* 形式，以表明我们正在求值一个调用，这与之前第3节基础语言语义中的情况一致。下一条规则处理要调用的对象不是组合器，而是挂起的符号查找（$/x/s$）的情况。在这种情况下，我们只需原样返回该调用，并标记为已尝试，因为无法继续推进。如果要调用的对象是另一个挂起调用（$/\text{atmdCall}\ x\ y/(T_1 \ldots)$），我们也采取相同做法。否则，该对象就是一个组合器，可能是原始组合器，也可能是派生组合器，我们从中提取包装_层级。然后，我们按包装_层级指定的次数，对参数依次进行部分求值、unval 和再次部分求值。如果组合器是普通的应用器，这意味着执行 1 次，就像大多数其他语言中的函数调用一样。另一种常见情况是执行 0 次，此时

$$[\text{peval}\ \text{freshCall}/(T_1\ T_2\ldots)\ /i_x/E\ ES\ FS] \quad\to\quad [\text{combine}[\text{peval}\ T_1\ /i_x/E\ ES\ FS](T_2\ldots)\ /i_x/E\ ES\ FS][\text{peval}\ \text{atmdCall}\ x\ y/(T_1\ T_2\ldots)\ /i_x/E\ ES\ FS] \quad\to\quad [\text{combine}[\text{peval}\ T_1\ /i_x/E\ ES\ FS](T_2\ldots)\ /i_x/E\ ES\ FS][\text{combine}\ /x/s\,(T_2\ldots)\ /i_x/E\ ES\ FS] \quad\to/\text{atmdCall}\ \emptyset\ \emptyset/(/x/s\ T_2\ldots)[\text{combine}\ /\text{atmdCall}\ x\ y/(T_1\ldots)\ (T_2\ldots)\ /i_x/E\ ES\ FS] \quad\to/\text{atmdCall}\ \emptyset\ \emptyset/\ (/\text{atmdCall}\ x\ y/(T_1\ldots)\ \ T_2\ldots)[\text{combine}\ (S\ n)\ \text{o}\ (V\ldots)\ /i_x/E\ ES\ FS] \quad\to\quad [\text{combine}\ n\ \text{o}[\text{peval}\ \text{unval}([\text{peval}\ V/i_x/E\ ES\ FS])\ /i_x/E\ ES\ FS]\ldots/i_x/E\ ES\ FS][\text{combine}\ /i''/\text{comb}\ (S\ n)\ \ s'\ /i'_{x'}/E'\ (s\ldots)\ \ Tb\ (V\ldots)\ /i_x/E\ ES\ FS] \quad\to\quad [\text{combine}\ /i''/\text{comb}\ \ n\ s'\ /i'_{x'}/E'\ s\ Tb[\text{peval}\ \text{unval}([\text{peval}\ \ V/i_x/E\ ES\ FS])\ /i_x/E\ ES\ FS]\ldots/i_x/E\ ES\ FS][\text{combine}\ /i''/\text{comb}\ \ 0\ s'\ /i'_{x'}/E'\ (s\ldots)\ \ Tb\ (V\ldots)\ /i_x/E\ ES\ FS] \quad\to\text{let}\ \ E'' = (s \leftarrow V)\ \ \ldots\ |s' \leftarrow /i_x/E,\ /i'_{x'}/E'\ \text{in let}\ \ C = /i''/\text{comb}\ \ 0\ s'\ /i'_{x'}/E'\ (s\ldots)\ \ Tb\ \text{in let}\ \ F = (C, E'')\ \text{in}\ /\text{atmdCall}\ \emptyset\ F/(C\ V\ldots)\ \text{if}\ \ F \in FS\text{else}\ \ [\text{under}[\text{peval}\ \ Tb/i''_r/E''\ (/i''_r/E'' \cup ES)\ \{F\}\ \cup FS](/i''/\text{comb}\ \ 0\ s'\ /i'_{x'}/E'\ (s\ldots)\ \ Tb\ \ V\ldots)\ /i_x/E\ ES\ FS][\text{under}\ \ z(/i''/\text{comb}\ \ 0\ s'\ /i'_{x'}/E'\ (s\ldots)\ \ Tb\ \ V\ldots)\ /i_x/E\ ES\ FS] \quad\to\text{let}\ \ o = returnOk(z, i')\ \text{in}dropRV(z, /i_x/E, ES, FS)\ \text{if}\ \ o = \text{true else let}\ \ i_? = i_x\ \text{if}\ s' \neq \emptyset\ \text{else}\ \ \emptyset\ \text{in}\ /\text{atmdCall}\ i_?\ \emptyset/(/i''/\text{comb}\ \ 0\ s'\ /i'_{x'}/E'\ (s\ldots)\ \ TbV\ldots)$$

图 11. 调用部分求值语义

该组合器是一个操作器，因此它会获取参数的代码而不对其求值。这样做可能是为了让操作器像宏一样工作，或实现特殊的控制流。wrap_level 大于 1 的数字也是可能的，但很少有用。为保持一致性，我们保留了 wrap_levels 大于 1 的能力，但在实践中并未使用。

现在，由于参数已根据组合器中的 wrap_level 求值了正确的次数，我们可以执行实际调用了。执行对原始组合器的调用，其语义以关系的形式列于附录 A 的图 9。对于派生组合器，我们会创建一个新的内部环境（E''），将参数符号映射到实参值，将特殊的动态环境参数符号（s'）映射到当前动态环境（$/i_x/E$），并从组合器值链接到指定的静态环境（$/i'_x/E'$）。我们首先检查当前是否正在执行这个（形式，环境）对（（C，E''）$\in FS$），如果是，则提前返回以防止无限递归。若确实提前返回，返回的调用会在其 *atmdCall* 形式中记录导致执行停止的（形式，环境）对：/atmdCall $\emptyset$ F/(C V . . .)。否则，会在这个新环境中对组合器函数体求值，并用*under*标记形式标记调用点。*under* 形式用于标示当前正在执行的调用位置。这样，我们就能在形式停止求值时，检查对调用点进行部分求值得到的值/代码是否合法、能否返回。为此，它既包含用于执行该调用的组合器函数体的部分求值结果，也包含一个回退项。本例中的回退项是一个更新后的挂起调用，在调用失败时使用。此外，回退项还携带当前动态环境、环境栈，以及供下一条规则使用的正在执行的形式集合。随后，使用辅助函数*returnOk* 检查该结果，判断此值能否安全返回（其定义见附录 A，概述见第 4.3 节）。如果可以安全返回，则在返回前将其传递给*dropRV*（附录 A 的图 8）。

如果存在或可能存在必须在此次调用创建的内部环境中解析的引用，那么返回值就是不合法的。如果结果可以合法返回，dropRedundentVeval（或 *dropRV*）会确保不会调用无用的 *veval*（在原语语义中使用的一种修改版 *eval*（见附录 A 中的图 9））来包装结果的任何部分。这一点很重要，因为在某些情况下，这些不必要的 veval 包装调用会阻碍部分求值的进展。

如果结果形式不合法，不能返回，我们就返回一个调用形式，其中包含部分求值后的组合器（更新后的 wrap_级别）以及根据 *under* 形式中的回退值部分求值后的参数。如果组合器确实接收动态环境，我们会在挂起调用形式中记录当前动态环境的标识符（$i_?$）。

4.3 辅助关系

接下来，我们简要介绍定义部分求值关系时用到的各种辅助关系（完整定义见附录 A）。

- **进展所需**：进展所需关系表示这样一组标识符：对于其中的每个标识符，至少需要有一个对应于真实环境（静态数据）的对象，且该环境中包含值，形式才能取得进展。true 表示无论具体情况如何，它都能取得进展。
- **进展所需上界**：进展所需关系的一个“上界”版本（见附录 A），它跟踪那些自身是真实环境、但向上链接到虚假环境标识符的环境的标识符。
- **进展所需无限递归**：该关系提取出先前已停止执行的形式，以防止无限递归。当我们当前不处于这些形式之一的调用栈中时，这一点很重要。如果不在其中，我们就应继续执行该形式，因为在遇到另一个可能导致无限递归的机会之前，它还有更多求值过程要完成。这是让我们能够使用 Y 组合子实现递归的机制之一，既不会导致无限递归，也不会在有限输入下留下尚未求值的递归调用。
- **查找**：顾名思义，它会在环境中查找与某个符号关联的值。

• **returnOk**：它用于判断能否将特定结果从特定环境中返回。例如，如果内部环境/组合器的 ID 没有显式出现在某个项中，也没有通过对接收其动态环境的组合器进行挂起调用而隐式出现，那么就可以从一次调用中返回该项。

• **IDin**：它用于判断此标识符是否出现在此值中，且未处于引入 thisID 的组合器之下。

• **takesDE**：它用于判断此组合器是否接收动态环境。
• **dropRV / dropRedundentVeval**：我们的最终辅助函数会移除多余的*veval*调用，这些调用可能妨碍进一步的部分求值。*veval* 是应用器*eval*的一个版本，但它的两个参数（要计算的项以及对其进行计算的环境）都已经 unval 并完成部分求值。它需要特殊处理（通过 -1 wrap_level 实现），因为它的项参数不应通过常规机制进行部分求值，否则会使用错误的环境（当前动态环境，而不是传递给*veval*/*eval*的显式环境）。*eval* 调用在类宏操作器中很常见，因为类宏操作器通常会在组合器函数体的最后调用*eval*，以计算在函数体执行过程中构造的代码。此*eval* 调用会部分求值为*veval*调用，随后便会成功返回到调用点。在常规的类宏操作器调用中，此调用点的动态环境就是传递给*veval* 的显式环境，因此*veval* 调用是多余的，项会直接内联，从而完成充分展开类宏操作器调用所需的部分求值过程。移除此调用是*dropRV*的职责。下文第 4.6 节中有一个“类宏操作符组合器调用”的示例，展示了这一过程。

4.4 部分求值原语的小步语义

最后，我们来看看原始组合子的（选定）语义。由于篇幅所限，它们的形式化定义放在附录 A 中。一般而言，它们执行的功能与较简单的基础原语相同，但作用对象是标记项。值得关注的主要组合子是 零判断 和 *vau*，它们分别对分支和函数体进行额外的部分求值。我们此前已经了解了 *eval* 的半衰期是怎样的：它会成为 *veval*，并在旁携带适当的环境，以便对其中一段暂停执行的代码求值，直到该代码被 *dropRV* 解封装，并插入最终位置。*eval*、*veval* 和 *dropRV* 的形式化定义见附录 A。省略的原语实现都很简单——只有当其参数是完全求值的值时才会求值，且求得的值与它们在基础语义下求得的值相同。

4.5 部分求值的综合效果

这套相互关联的机制最终消除了所有静态调用的操作器，其中操作器是以类似宏的风格编写的。请注意，部分求值所维持的不变量确保，只有参数全为值的派生组合器才会执行。此外，对组合器的调用所返回的结果，要么是一个值，要么是一个挂起计算，其中所有调用要么不接收动态环境（也就是它所返回自的组合器的环境），要么显式调用 veval 并提供自己的环境。这些都是类宏组合器可以返回的情形！此外，部分求值过程会剔除冗余的 veval 调用。从类宏组合器的调用中返回挂起代码后，其中对 veval 的调用就会变得多余，内部的挂起计算也会内联到父级中，就像宏会被展开为代码并插入其调用位置一样。

4.6 示例

为说明该算法，我们提供了一些分阶段进行标记、取消求值和部分求值的示例。逐步完整示例对于任何论文来说都太长，因此这里做了一些删节。

4.6.1 加法示例。 我们将完整演示 (+ 1 2)（值为 3）的部分求值步骤。

(+ 1 2) /值/(/∅/+ 1 2) /freshCall/(/true/+ 1 2)	初始代码 已标记 然后取消值化
[peval /freshCall/(/true/+ 1 2) /i_r/*E ES FS*]	我们无法展示整个环境， 但为了说明，假设 E 将“+”映射到原始组合器 + combiner
[combine [peval /true/+/i_r/*E ES FS*](1 2) /i_r/*E ES FS*] [combine 1 + (1 2) /i_r/*E ES FS*]	开始调用，PV 组合子 查找会将符号 + 替换为 原始组合器 组合器
[combine 0 + ([peval unval(1) /i_r/*E ES FS*] [peval unval(2) /i_r/*E ES FS*]) /i_r/*E ES FS*]	Unval+部分求值以进行求值 参数，但整数仍然是 same
[combine 0 + (1 2) /i_r/*E ES FS*] +(1 2) 3	然后调用 原始组合器进行计算 结果是 3，正如预期， 可以合法地返回

4.6.2 常量组合器示例。 完成加法之后，我们将通过引入一个组合器的创建来稍微增加复杂度。这个组合器的函数体可以进行部分求值。由于我们已经详细讲解过简单加法，因此这里会迈出更大的步子。在本例中，我们有 (*vau* (*x*) (+ 1 2 *x*))，这表示对于某个输入 x，得到“1+2+x”。我们只用了一点语法糖，让一个双参数 vau 等价于一个忽略其特殊动态环境参数的三参数 vau。

(*vau* (*x*) (+ 1 2 *x*)) /值/(/∅/*vau* /值/(*x*) /值/(/∅/+ 1 2 *x*)) /freshCall/(/true/*vau* /值/(*x*) /值/(/∅/+ 1 2 *x*))	初始代码 已解析并标记的语法 已取消值化
[peval /freshCall/ (/true/*vau* /值/(*x*) /值/(/∅/+ 1 2 *x*)) /i_r/*E ES FS*]	我们无法展示整个环境。 举例来说，E 将 "vau" 映射 到原始 vau 组合子
[combine [peval /真/*vau* /i_r/*E ES FS*] (/值/(*x*) /值/(/∅/+ 1 2 *x*))) /i_r/*E ES FS*]	开始调用，PV 组合子
[peval /7/**comb** 0 *s′* /i_f/*E* (*x*) /freshCall/(/true/+ 1 2 /∅/*x*)	符号 Vau 映射到其 组合器值，现在将

$/i_r/E\ ES\ FS]$ 被部分求值

$/7/\mathbf{comb}\ 0\ s'\ /i_f/E\,(x)$ 对函数体进行部分求值
[peval /freshCall/(/true/+ 1 2 $/\emptyset/x$) 使用伪环境。请注意，我们
$/7_f/(x \leftarrow /7/x)\ \rfloor,\ /i_r/E\ \ ES\ FS]$ 几乎回到了第一个示例

$/7/\mathbf{comb}\ 0\ s'\ /i_r/E\,(x)$ /atmdCall $\emptyset\ \emptyset/(0 + 3\ /7/x)$ 我们将快速跳过
第一个示例中的过程

我们快速略过了与前一个示例大致相同的部分。唯一的区别是在加法中加入了参数引用 x。需要注意的是，$/\emptyset/x$先被 unval 成 /true/x（未显示），随后又被部分求值为 /7/x（7 是组合器／伪环境的标识符）。此外，加法的部分求值版本必须返回一个新的部分求值调用，因为它还无法对 x求值。这样我们就得到了一个部分求值过的组合器，但目前还没看到这种新技术能做到以往的部分求值技术做不到的事。为此，我们来看看类宏操作调用如何被部分求值消去的高层示意。

4.6.3 类宏操作调用示例。 由于逐步求值这段代码的过程会很长，我们将比之前跳得更大一些，省略那些可以从前两个示例中推断出的内容。

```
1 (let ( ( double_parameter (vau de (x) (eval ( 数组  +x x) de))) ) 2 (vau (x) (double_
pa
```

我们暂且跳过“let”的工作原理（它也是一个操作子组合器），重点关注类宏操作器部分求值。这段代码定义了一个名为“double_parameter”的类宏操作器，它接收一段尚未求值的代码“x”以及动态调用环境“de”，然后将代码“(+ x x)”构造成一个数组，并在调用环境 de 中对其求值。这段代码刻意写得很简单，会将参数求值两次（不过，在纯函数式语言中，求值 x 不会产生副作用，因此这样做的缺点会大大减轻）。这应该会让你想到 C语言中的经典宏示例“#define double(x) (x+x)”，我们选择它确实是因为它广为人知。我们先来看看这个类宏 f-表达式部分求值后的样子，然后直接进入它的应用。代码如下：

```
1 (vau de (x) (eval ( 数组  +x x) de))
```

变为

$/6/\langle\mathbf{comb}\ 0\ \ de/i_r/E(x)$ /atmdCal $\emptyset\ \emptyset/(0\ \mathbf{eval}$ /atmdCal $\emptyset\ \emptyset/(0\ \mathbf{数组}\ 1 + /6/x/6/x)\ /6/de\ \ )\rangle$

与我们先前关于部分求值组合器的示例非常相似。我们有一个组合器，其中有两个嵌套的挂起调用，以及一些挂起的符号查找。

现在让我们看看它在部分求值函数体时的用法：

```
1 (vau (x) ( double_参数 (+  1  2  x)))
```

变为……，然后向前跳过，开始求值函数体

$/7/\langle\mathbf{comb}\ 0\ \emptyset\ /i_f/E\ (x)$ [peval /freshCall/(/true/*double_parameter* /val/(/$\emptyset$/+ 1 2 /$\emptyset$/x)) $/i_r/\langle\langle(x \leftarrow /7/x)\ \rfloor,\ /i_r/E\rangle\rangle\ ES\ FS]\rangle$ 转发至调用

$$/7/\langle \textbf{comb}\ 0\ \emptyset\ /i_f/E\ (x)\ [\text{peval}\ /\text{freshCall}/(\\ /6/\langle \textbf{comb}\ 0\ de\ /i_r/E\ (x)\ /\text{atmdCall}\ \emptyset\ \emptyset/\\ (\langle 0\ \textbf{eval}\rangle\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ \textbf{array}\rangle\ \langle 1\ +\rangle\ /6/x\ /6/x)\ /6/de)\rangle\\ /\text{val}/(/\emptyset/+\ 1\ 2\ /\emptyset/x))\\ /7_r/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle\ ES\ FS]\rangle \qquad \text{substituting in definition}$$

我们将用一个符号来表示这个嵌套环境，因为它相当繁琐。

$$E'' = /6_r/\langle\langle (x \leftarrow /\text{val}/(/\emptyset/+\ 1\ 2\ /\emptyset/x))\ |/7_r/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle,\ /i_r/E\rangle\rangle$$

让我们继续看这个示例。

$$[\text{peval}\ /\text{atmdCall}\ \emptyset\ \emptyset/ \qquad \text{evaluating body}\\ (\langle 0\ \textbf{eval}\rangle\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ \textbf{array}\rangle\ \langle 1\ +\rangle\ /6/x\ /6/x)\ /6/de)\ E''\\ ES\ FS] \qquad \text{with generated env}$$

$$[\text{peval}\ /\text{atmdCall}\ \emptyset\ \emptyset/ \qquad \text{evaluating array call}\\ (\langle 0\ \textbf{eval}\rangle\ /\text{val}/(\langle 1\ +\rangle\ (/\emptyset/+\ 1\ 2\ /\emptyset/x)(/\emptyset/+\ 1\ 2\ /\emptyset/x))\\ /7_f/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle)\ E''\\ ES\ FS] \qquad \text{and its symbol parameters}$$

$$[\text{combine}\ /i''/\langle -1\ \textbf{veval}\rangle\\ (\text{unval}(/\text{val}/(\langle 1\ +\rangle\ (/\emptyset/+\ 1\ 2\ /\emptyset/x)(/\emptyset/+\ 1\ 2\ /\emptyset/x)))\\ /7_f/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle)\\ E''\ ES\ FS] \qquad \text{evaluating eval}$$

$$[\text{combine}\ /i''/\langle -1\ \textbf{veval}\rangle\\ (/\text{freshCall}/(\langle 1\ +\rangle\ (/\emptyset/+\ 1\ 2\ /\emptyset/x)(/\emptyset/+\ 1\ 2\ /\emptyset/x))\\ /7_f/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle)\\ E''\ ES\ FS] \qquad \text{unval}$$

$$\text{let}\ /7_f/E' = /7_f/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle\ \text{in} \qquad \text{we also abbriviate E'}\\ \text{let}\ V' = [\text{peval}\\ /\text{freshCall}/(\langle 1\ +\rangle\ (/\emptyset/+\ 1\ 2\ /\emptyset/x)(/\emptyset/+\ 1\ 2\ /\emptyset/x))\\ /7_f/E'\ \{/7_f/E'\} \cup ES\ FS]\ \text{in}\\ [\text{under}\ V'\ (\langle -1\ \textbf{veval}\rangle\ V'\ /7_f/E')\ /i_x/E\ ES\ FS] \qquad \text{stepping veval}$$

V' 将变为：

$$\text{let}\ V' = /\text{atmdCall}\ \emptyset\ \emptyset/(0 + /\text{atmdCall}\ \emptyset\ \emptyset/(0 + 3\ /7/x)\ /\text{atmdCall}\ \emptyset\ \emptyset/(0 + 3\ /7/x))\ \text{中}$$

现在，内部调用将无法完成，因为 *returnOk* 会返回 false，结果中找到了 7。因此，将返回回退值，并作为调用 *double_parameter* 的最终结果。这一次，*returnOk* 会为 true，因为结果中找不到 6（*double_parameter*的标识符）。因此，我们将转而调用 *dropRV*，其标识符为 7。

$$dropRV(/\text{atmdCall}\ \emptyset\ \emptyset/(\langle -1\ \textbf{veval}\rangle\ V'\ /7_f/E'),\ /7_f/E',\ ES,\ FS)$$

当然，这是一次冗余的 veval，因此对调用*double_parameter* 进行部分求值后的最终结果是：

$$V'$$

也就是说，

$$/\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ + \rangle\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ + \rangle\ 3\ /7/x)\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ + \rangle\ 3\ /7/x))$$

请注意，这个挂起计算会被返回，并取代对类宏操作器的调用。这段代码等价于：

```
1 (vau (x) (+ (+ 3 x) (+ 3 x)))
```

This 这就是我们使用宏和基本常量传播时会得到的结果 n.

请注意，对这个示例的求值并不依赖于具体的值、类宏操作器生成的代码，或调用的参数。正是如此，我们的算法会以一种与更常见的优化型 Scheme 实现中先进行宏展开、再进行常量传播相一致的方式，将对类宏操作器的静态调用部分求值掉。

5 编译器后端和优化

部分求值可以消除类宏操作器带来的所有低效问题，但仍有其他低效问题需要通过后端优化来解决。一个主要的剩余低效问题是动态组合器调用点。在这种情况下，事先没有可用的局部信息来确定将被调用的组合器是应用器还是操作器——也就是说，参数是否会被求值，以及组合器是否需要访问其调用动态环境。这通常意味着，动态调用中位于参数位置的代码甚至无法进行部分求值。为克服这一低效问题，我们在编译后的组合器闭包值上标记一些比特，以指示其包装层级以及是否需要调用环境。每个动态调用点都会根据这些比特进行分支。在动态分支的“wrap_level=0”一侧，会生成一个对静态内存中写出的未求值参数的引用。在动态分支的“wrap_level=1”一侧，编译器会重新调用 unval，并对每个参数递归地执行部分求值器，从而生成效率再次媲美不含 fexpr 的语言的代码（外加动态分支的开销）。请注意，这要求扩展上述部分求值算法以及编译算法，以支持失败。部分求值期间失败并不一定意味着运行时失败。例如，动态组合器可能始终是一个包装层级为 0 的操作器，因此某些会报错的参数代码实际上是数据，永远不会被求值（因而也永远不会导致错误）。

为求简单，垃圾回收采用引用计数。由于这是一种纯函数式语言，因此无需担心循环引用。为了消除部分求值后剩余的低效问题，我们实现了多种编译器优化。以下各节将介绍我们在编译器中实现的其他关键优化。

5.1 惰性环境实例化

我们会延迟分配和初始化动态环境值，直到确实需要时才进行。接收动态环境的组合器调用会检查专用寄存器，以确定环境值是否已创建。如果尚未创建，就会创建它。这意味着，对于那些没有任何调用会接收动态环境的组合器调用，其动态执行轨迹永远不会将动态环境具体化，并且只会产生一次可预测的分支开销。对于静态组合器调用，这些信息（例如是否对参数求值、是否接收外围环境等）在编译时已知，因此不会生成运行时分支。

5.2 基于类型推断的原语内联

为了减少每个内置操作都作为动态类型组合器调用所带来的开销，我们实现了原始操作的类型推断引导内联。分析遍会根据分支谓词推断类型；对于匹配操作器生成的代码，这种方法效果相当好。例如，通过确定某个变量在 *cond* 中必须包含一个数组，可以将组合器 长度 内联为仅由几条位操作码组成的代码。

例如，请看以下代码：

```
(cond   (and (数组？ a) (= 3 (长度 a)))      (idx a 2)
        true                                 nil)
```

清单 5. 类型推断示例

对 *idx* 的调用可以完全内联，无需进行类型检查或边界检查，因为它位于一个代码块中，而只有在变量 'a' 确实包含长度为 3 的数组时才可能到达该代码块。内联类型谓词不需要类型信息，因为它们只需检查标签位。如果任何参数属于可以进行字/指针比较的类型（整数、布尔值和符号），就可以将相等性检查内联为简单的字/指针比较。类型推断与原语内联结合后，意味着在大多数匹配表达式中，每个原始组合器调用都可以完全内联为几个操作码。在上述示例中，列出的每个原始组合器都会内联：*cond* 会内联为 WebAssembly 的 if 块，谓词函数会内联为位操作和分支，idx 会内联为位操作以及带常量偏移量的加载，等等。

5.3 立即调用的闭包内联

内联对闭包值的调用——这些闭包值在分配后立即使用——有助于在实现某些操作器时不产生额外开销。受益最大的类宏操作器是“let”。如下所示，清单 6 经过部分求值后等价于清单 7，然后再进行内联。因此，唯一的开销是创建一个新环境；而通过惰性环境实例化，这一过程会进一步变为惰性，并在常见情况下被消除。由此可见，“let”实际上是定义一个闭包并立即调用它的语法糖，这与许多 λ 演算中的情况一样，但这样做不会损失效率。

```
    (let (a (+ 1 2))
         (+ a 3))
```

列表 6. Let 内联示例

```
    ((包装 (vau (a) (+ a 3))) (+ 1 2))
```

列表 7. Let 内联示例 - 展开

5.4 Y 组合子消除

延续将经典 λ演算概念的实现做得与标准实现同样高效这一主题，最后一组优化确保使用 Y 组合子实现递归时不会产生任何额外开销。在 Kraken 中，Y 组合子如清单8 所示，其用法的一个简单示例如清单9所示。

```
(let Y  (lambda (f)
            ((lambda (x) (x x))
             (lambda (x) (f (wrap (vau app_env (& y) (lapply (x x) y app_env)))))))
)
```

清单 8. Kraken 中定义的 Y 组合子

```
(Y (lambda (递归) (lambda (n) (if (= 0 n) 1
                                  (* n (递归 (- n 1)))))))
```

清单 9. 显式使用 Y 组合子的阶乘函数

通常，人们不会手动使用 Y 组合子。在 Kraken 中，有一个易于使用的递归 *lambda* 派生操作器，它会在幕后求值 Y 组合子。无论是否明确声明，Kraken 实际上始终使用 Y 组合子来实现递归。这样我们就能保持语言的纯粹性，并使其符合演算。

这种优化实际上是我们架构自然带来的，只需在记录过程中稍加留意即可。编译组合器时，编译器会先将组合器的索引写入记忆化字典，然后再对组合器的函数体重新执行部分求值。任何静态递归调用的形式都与当前正在编译的组合器完全一致，因此编译器可以生成指向正确组合器索引的静态引用。之所以可行，是因为在编译之前对函数体重新执行部分求值，确保了递归调用中的组合器形式被规范化为与当前正在编译的组合器相同。由于这是一种急切求值语言，我们语言中的 Y 组合子定义额外包含一个闭包，以防止 Y 组合子自身发生无限递归。因此，我们还在编译器中实现了 η-转换，以移除这一额外的间接层。由于内部表达式现在是一个常量，而不是一次调用，因此不会有无限递归的风险。结合上述规范化处理，使用 Y 组合子定义递归函数时，我们就能实现完全高效的静态递归调用。

最后，作为一种纯函数式 Lisp，我们使用递归而非迭代。在等待 WebAssembly 合并并实现 tail_call 指令期间，我们实现了一种功能较有限的尾调用消除：将尾位置上的自递归调用转换为跳转到循环起始处的分支，而该循环包围着组合器的函数体。将尾调用消除与上文的 Y 组合子消除结合起来，意味着使用 Y 组合子定义的递归函数可以像其他语言中的命令式循环一样高效。WebAssembly 完成 tail_call 指令的实现后，我们便可以轻松生成该指令，从而实现完整的正确尾调用。

6 基准测试与评估

我们对*Kraken* 进行评估，以回答以下问题：

- 部分求值和我们实现的编译器优化能在多大程度上提升*Kraken*的性能?
- 与其他 fexpr 实现相比，我们的纯函数式 f-expr 语言*Kraken* 快多少?
- 带有 fexpr 的*Kraken*，其性能与宏相比如何?
- Kraken 中不同的部分求值机制/优化如何降低整体运行时?

实验设置：我们在一台安装了 NixOS 的设备上，通过可复现的 Nix 环境运行了这些实验 [Dolstra 和 Löh 2008]（内核 6.0.0），该设备配备 8 个核心 / 16 个线程和 64 GB 内存。我们的代码包含用于复现测试所需的确切依赖项的脚本和 Nix Flakes。

Kraken 基准测试中的大多数测试都分别使用 Wasmtime 和 WAVM 这两种 WebAssembly 引擎运行。Wasmtime WebAssembly 引擎由 Bytecode Alliance 自行开发，是最受欢迎的引擎之一，并使用 CraneLift 代码生成后端。WAVM WebAssembly 引擎因采用 LLVM 而颇具特色；它通常能在基准测试中生成最快的代码，但启动时间较长。由于运行时会耗尽栈空间（这是 Cfold 基准测试已知的特性），我们移除了 Cfold Wasmtime 基准测试。

基准测试：为了展示 Kraken 的能力，我们创建了这些基准测试。它们通常以函数式语言实现，也曾被其他论文用作基准测试 [Reinking 等，2021； Westrick 等 2022]。这些基准测试包括

- 斐波那契 - 计算第 n 个斐波那契数
- 红黑树 - 向红黑树中插入 n 个项目，然后遍历该树并对其中的值求和
- Deriv - 计算大型表达式的符号导数
- 常量折叠 - 对大型表达式进行常量折叠
- N 皇后问题 - 在棋盘上放置 n 个皇后，使任意两个皇后之间都不处于同一对角线、同一列或同一行

除斐波那契以外，所有基准测试都使用 *Kraken* 中用于模式匹配的 fexpr 版本的 match，这与 NewLisp 中的宏版本等效。我们还使用 NewLisp 的 [Mueller2018] 版本的 fexpr match 对红黑树进行基准测试。我们调整了基准测试所用问题的规模，以适应某些优化程度较低的实现运行时间较长的情况。Kraken 和 NewLisp 的代码非常相似，需要指出的是，这种写法非常不符合 NewLisp 的惯用风格。我们的目标并非比较 Kraken 和 NewLisp 作为红黑树实现语言的优劣，而是对一个相当复杂的 fexpr/宏进行压力测试，即模式匹配。

6.1 部分求值对 eval 调用的影响

表 5. F 表达式未进行部分求值时的 eval 调用次数

	求值次数	Eval w1 调用次数	Eval w0 调用次数	编译动态调用 w1 调用次数	编译动态调用 w0 调用次数
常量折叠 5	10897376	2784275	879066	1	0
Deriv 2	11708558	2990090	946500	1	0
N 皇后问题 7	13530241	3429161	1108393	1	0
斐波那契 30	119107888	30450112	10770217	1	0
红黑树 10	5032297	1291489	398104	1	0

如前所述，不进行部分求值就使用 fexpr，会妨碍前置优化，并导致大量重复工作。表 5 和表6 展示了以下调用次数：调用*Kraken* 运行时的 eval 函数、运行时的 eval 函数以包装层级_1=1调用应用器的次数、运行时的 eval 函数以包装层级_0=0调用操作器的次数、编译的动态调用包装层级为_1=1的应用器的次数，以及编译的动态调用包装层级为_0=0的操作器的次数。这些数据分别展示了 *Kraken* 测试用例关闭和开启部分求值时的情况。

表 6. 部分求值的 f 表达式中的 eval 调用次数

	求值次数	eval w1 调用次数	eval w0调用次数	编译动态调用 w1 调用次数	编译动态调用 w0 调用次数
常量折叠 5	0	0	0	0	0
Deriv 2	0	0	0	2	0
N 皇后问题 7	0	0	0	0	0
斐波那契 30	0	0	0	0	0
红黑树 10	0	0	0	10	0

表 7. 红黑树运行时 eval 函数的调用次数。表中显示了非部分求值的数字 ->部分求值的数字。

	求值次数	求值 w1 调用次数	求值w0 调用次数	编译动态调用 w1 调用次数	编译动态调用 w0 调用次数
红黑树 7	2952848 -> 0	757932 -> 0	233513 -> 0	1 -> 7	0 -> 0
红黑树 8	3532131 -> 0	906548 -> 0	279379 -> 0	1 -> 8	0 -> 0
红黑树 9	4278001 -> 0	1097965 -> 0	3383831 -> 0	1 -> 9	0 -> 0

不进行部分求值，就无法进行编译，因为无法判断调用的参数是否会被求值。在所有基准测试中，部分求值都移除了对运行时 eval 函数的所有调用，从而生成了完全编译的程序。以红黑树为例，对包装层级为 1 的组合器（普通函数）的调用超过一百万次，对包装层级为 0 的组合器（替代宏的操作器）的调用有 398,000 次。调用次数之所以大幅增加，是因为在其他类宏组合器的定义中，类宏组合器被反复以指数级的方式重新执行，正如引言中所讨论的那样。

未经部分求值的基准测试显示，有 1 次对应用器的编译动态调用（这是首次调用 eval），以及 0 次对操作器的编译动态调用，因为完全没有进行编译。对于经过部分求值的基准测试，由于其中使用了高阶函数，因此有少数对应用器的编译动态调用；而对操作器的编译动态调用则为零，因为所有操作器的使用都已消除。我们还改变了红黑树的输入规模，如表 7 所示，以展示这些数字如何随输入规模变化。

这些表格所显示的惊人性能差距，在图 13中的红黑树测试里体现得淋漓尽致。我们缩短了这次测试，因为 Kraken 未进行部分求值的解释器即使处理 100 次插入，也要耗费极长时间（40 分钟）。在未经部分求值的 Kraken 版本中，层层重复的类宏操作器调用相互叠加，导致其速度比经过部分求值、优化和编译的版本慢 70,000 倍。对于其余基准测试，我们移除了朴素的解释执行版 *Kraken* ，因为它在每种情况下的性能都糟糕到会撑爆图表，令比较无法进行。在优化后的 Kraken 中，我们的部分求值算法能够彻底消除这些低效层级，对结果求值并将其内联，同时为后端提供更专门化的代码进行优化，生成的编译版本轻松胜过 NewLisp-fexpr 实现，甚至也胜过 NewLisp-macro 实现，如图 14所示。本次测试中，我们将基准测试规模设得较小，因为 NewLisp 的堆栈限制使规模无法超过 880；而 *Kraken* 编译器执行的尾调用消除使我们能够运行规模大得多的基准测试，包括向红黑树插入 4,800,000 个元素的测试。这一结果展现了部分求值和编译器优化对 *Kraken*运行时性能的显著影响。我们的技术将完全基于 F 表达式的语言从完全无法实用，提升到速度快于目前正在使用的基于宏的动态脚本语言。

6.2 Kraken 版本之间的比较

除了部分求值带来的巨大提速之外，图 12和13 展示了逐一禁用我们所述的各种编译器优化后产生的影响。我们的四项主要优化对运行时有显著的正面作用，惰性环境实例化除外。惰性环境实例化能大幅提升斐波那契基准测试的表现，也能在一定程度上改善 Deriv 的表现，但通常会使其余部分略微变慢。

图 12. 常量折叠和 Deriv

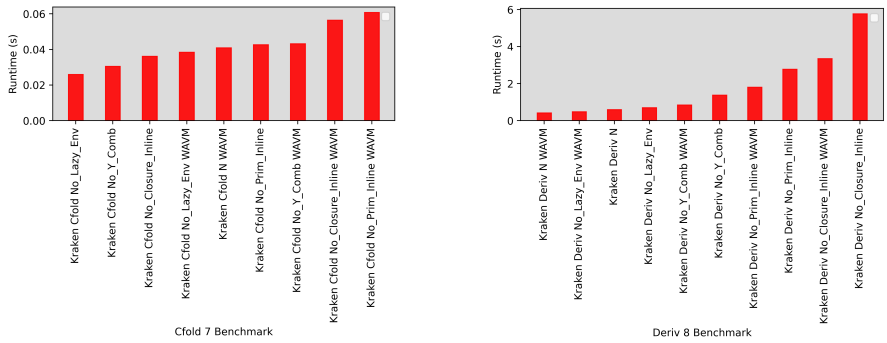
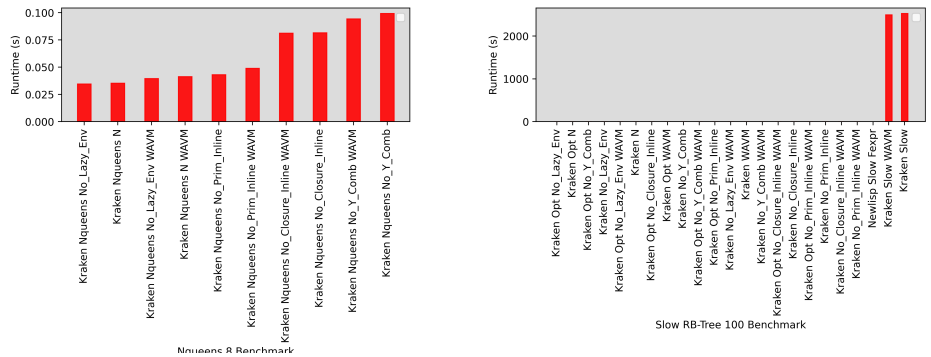


图 13. N 皇后问题



6.3 与其他方法的比较

为了大致展示当前的性能，我们还展示了一个斐波那契基准测试，它主要考察纯函数调用速度和内联，如图 14所示。我们加入了 Python 和 Chez Scheme，以大致说明典型的慢速和快速动态语言分别处于什么水平。得益于部分求值、编译以及成熟的 WebAssembly 实现，我们的表现胜过了二者；但对此结果应谨慎看待，因为这只是一个非常有限的微基准测试，其目的仅在于大致展示我们性能的数量级。

7 相关工作

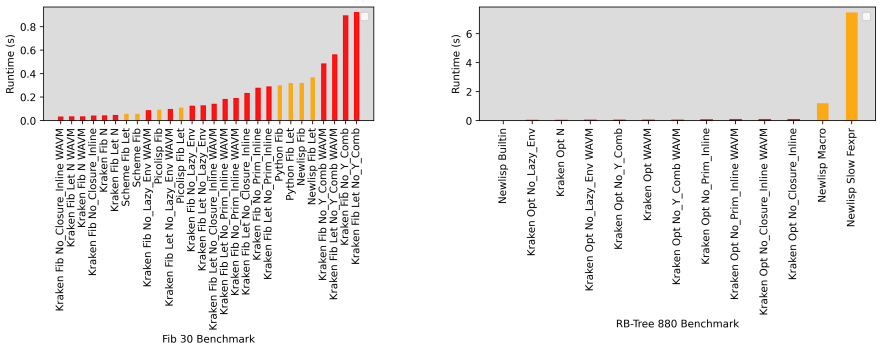
本文重点探讨如何利用部分求值高效实现 fexpr 语言。

其他 F 表达式实现：

除了 NewLisp，[Mueller 2018], picoLisp [Burger 2013] 也将 fexpr 作为一种语言特性。与 NewLisp 类似，picoLisp 是一种动态解释型语言，不会对 fexpr 进行特殊优化。与 NewLisp 一样，picoLisp 中的大多数形式都是直接在解释器中实现的，而不是由较小的组件组合而成；在使用宏的其他 Lisp 方言中，宏通常就是这样构建的。这两种语言最初都没有宏实现，但 NewLisp 后来增添了宏，因为它的 fexpr 导致了令人困惑的求值规则和性能问题。

然而，John Shutt 在他 2010年 的论文中证明 [Shutt 2010] 如果将 fexpr 从其历史背景以及与其他如今已不复存在的语言特性的共存状态中剥离出来，重新进行形式化定义，

图 14. Kraken 与其他方案的比较。按速度从快到慢排序



那么它们就可以成为动态作用域等特性的优雅且行为良好的宏替代方案。不过，语言从一开始就必须围绕这一点进行设计。我们还证明了长期困扰 fexpr 的性能问题是可以解决的，从而解决了阻碍 fexpr 广泛应用的另一项主要问题。

部分求值：

我们工作的主要贡献是提出了一种部分求值方案，为提升 fexpr 实现的性能奠定了基础。部分求值技术已有多年历史，早在 Lisp 的早期发展阶段就已出现 [Lombardi 1964], , 并且已应用于许多语言。Jones [Jones 1996] 以及 Charles 与 Olivier [Consel 和 Danvy 1993] 从较高层次介绍了部分求值及其在相关领域的发展历史，并清晰地阐释了在线技术与离线技术之间的差异。Ruf 在其论文中详细介绍了一些有趣的在线部分求值技术 [Ruf 1993], ; Sperber 与 Thiemann [Sperber and Thiemann 1996] 在关于使用部分求值进行编译的论文中，描述了以静态数据描述代替动态数据的技术，这与我们处理部分静态数据环境的方式相似。Danvy、Malmkjær 和 Palsberg 在实现“The Trick”时将 η 扩展用作部分求值的一环 [Danvy et al. 1996], ; 而我们则在编译期间使用它来优化 Y组合子 的残余部分。部分求值应用广泛，已用于许多命令式语言，例如 C语言 [Andersen 1992], MATLAB [Elphick et al. 2003] 和 Pascal [Meyer 1991], , 也用于其他编程范式，例如逻辑编程。Lloyd 与 Sheperdson [Lloyd and Shepherdson 1991]为此奠定了理论基础，Alpuente 等人随后对其进行了扩展 [Alpuente et al. 1998]。Romph 等人使用部分求值／分阶段编程来优化 Scala 中的数据结构 [Rompf et al. 2013], 。Sperber 与 Thiemann 还将部分求值用于生成 LR分析器 [Sperber and Thiemann 2000],; Sestoft 则使用部分求值来编译高效的 ML 模式匹配 [Sestoft 1996]。

8 结论与未来工作

在这项工作中，我们提出了一种基于 fexpr 的纯函数式 Lisp，*Kraken*，并首次提出一种编译框架，通过利用部分求值使 fexpr 能够高效运行。部分求值器会消除所有表现得像宏的 fexpr（即特定形式的操作器），这表明，我们类似宏的 fexpr 效率可以媲美编译时宏，而动态函数调用只会受到轻微影响。此外，部分求值器与编译器优化生成的代码，比基础解释器（快 70,000 倍）或 NewLisp 的 fexpr 实现更优化 [Mueller 2018]（快 233 倍）。我们已经证明，fexpr 可以是一种性能相当不错的

宏的替代方案，也是构建 Lisp 传统中富有表现力的新型函数式语言的可行基础。

未来，我们将研究如何改进部分求值器，使其能够处理定义方式更加灵活、而不那么严格类似于宏的操作器；同时还将探讨添加定界续延等特性的可能性。为简化起见，我们曾从 Shutt的Vau演算中移除了这些特性。

参考文献

María Alpuente、Moreno Falaschi 和 Germán Vidal。1998年。函数逻辑程序的部分求值。《ACM〈编程语言与系统汇刊〉》（TOPLAS）20，4（1998年），768–844。Lars Ole Andersen。1992年。《自适用C程序特化》。PEPM 92，28（1992年），54–61。Andrew Berlin。1990年。将部分求值应用于数值计算。载于《1990年ACM Lisp与函数式编程会议论文集》。139–150。Andrew Berlin 和 Daniel Weise。1990年。使用部分求值编译科学代码。《Computer》23，12（1990年），25–37。Alexander Burger。2013年。《PicoLisp参考手册》。William Clinger 和 Jonathan Rees。1991年。行之有效的宏。载于《第18届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集》。155–162。Charles Consel 和 Olivier Danvy。1993年。《部分求值教程讲义》。载于《第20届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集》（美国南卡罗来纳州查尔斯顿）（POPL '93）。计算机协会，纽约州纽约市，美国，493–501。https://doi.org/10.1145/158511.158707 Olivier Danvy、Karoline Malmkjær 和 Jens Palsberg。1996年。η扩展妙用。《ACM〈编程语言与系统汇刊〉》（TOPLAS）18，6（1996年），730–751。Eelco Dolstra 和 Andres Löh。2008年。NixOS：纯函数式 Linux 发行版。SIGPLAN通讯 43，9（2008年9月），367–378。https://doi.org/10.1145/1411203.1411255 Daniel Elphick、Michael Leuschel 和 Simon Cox。2003年。MATLAB的部分求值。载于《生成式编程与组件工程国际会议》。施普林格，344–363。Matthew Flatt 和 PLT。2010年。参考手册：Racket。技术报告 PLT-TR-2010-1。PLT Design Inc. https://racket-lang.org/tr1/。Yoshihiko Futamura。1971年。计算过程的部分求值——一种编译器生成器方法。《Systems, computers, controls》25（1971年），45–50。Rich Hickey。2008年。Clojure编程语言。载于《2008年动态语言研讨会论文集》。1–1。Neil D. Jones。1996年。《部分求值导论》。ACM《计算调查》28，3（1996年9月），480–503。https://doi.org/10. 1145/243439.243447 Logan Kearsley。 [无日期]。《实现一种支持多种求值策略的基于Vau的语言》。([无日期])。H. Jan Komorowski。1982年。部分求值作为应用型语言中数据结构推断的手段：理论及其在Prolog中的实现。载于《第9届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集》（新墨西哥州阿尔伯克基）（POPL '82）。计算机协会，纽约州纽约市，美国，255–267。https://doi.org/10.1145/582153.582181 J.W. Lloyd 和 J.C. Shepherdson。1991年。逻辑编程中的部分求值。《逻辑编程杂志》11，3（1991年），217–242。Lionello A Lombardi。1964年。Lisp作为增量计算机的语言。（1964年）。Uwe Meyer。1991年。命令式语言的部分求值技术。载于《1991年ACM SIGPLAN部分求值与基于语义的程序变换研讨会论文集》。94–105。Lutz Mueller。2018年。NewLisp。Alex Reinking、Ningning Xie、Leonardo de Moura 和 Daan Leijen。2021年。Perceus：无垃圾回收的引用计数与复用。载于《第42届ACM SIGPLAN编程语言设计与实现国际会议论文集》。96–111。Tiark Rompf、Arvind K Sujeeth、Nada Amin、Kevin J Brown、Vojin Jovanovic、HyoukJoong Lee、Manohar Jonnalagedda、Kunle Olukotun 和 Martin Odersky。2013 年。优化高级程序中的数据结构：基于分阶段编程的可扩展编译器新方向。载于《第40届ACM SIGPLAN-SIGACT编程语言原理年度研讨会论文集》。497–510。Erik Steven Ruf。1993年。《在线部分求值专题》。斯坦福大学。Peter Sestoft。1996年。ML模式匹配编译与部分求值。载于《Dagstuhl部分求值研讨会》，第1110卷。446–464。John N Shutt。2010年。《F表达式作为Lisp函数应用的基础，或 $vau：终极抽象》。博士论文。Michael Sperber 和 Peter Thiemann。1996年。通过部分求值实现现实编译。SIGPLAN通讯 31，5（1996年5月），206–214。https://doi.org/10.1145/249069.231419

Michael Sperber 和 Peter Thiemann。2000年。通过部分求值生成 LR 分析器。*ACM*《编程语言与系统汇刊》（*TOPLAS*）22，2（2000年），224–264。Sam Westrick、Jatin Arora 和 Umut A. Acar。2022年。近乎零成本的纠缠检测。《*ACM* 程序语言会议录》6，ICFP，文章115（2022年8月），32页。https://doi.org/10.1145/3547646 Patrick Henry Winston 和 Berthold K Horn。1986年。Lisp。（1986年）。Thomas Würthinger、Christian Wimmer、Christian Humer、Andreas Wöß、Lukas Stadler、Chris Seaton、Gilles Duboscq、Doug Simon 和 Matthias Grimmer。2017年。面向高性能动态语言运行时的实用部分求值。*SIGPLAN*通讯52，6（2017年6月），662–676。https://doi.org/10.1145/3140587.3062381

A 辅助关系

$$\begin{aligned}
&\mathsf{neededIDs}(n) = \quad \emptyset \quad \mathsf{neededIDs}(/\emptyset/s) = \quad \emptyset \quad \mathsf{neededIDs}(/i/s) = \{i\} \quad \mathsf{neededIDs}(/\text{值}/(T_1 \ \ldots \ T_n)) \\
&= \mathsf{neededIDs}(T_1) \ \cup \cdot \cdot \ \cup \mathsf{neededIDs}(T_n) \quad \mathsf{neededIDs}(/\mathsf{freshCall}/(T_1 \ \ldots \ T_n)) = \{\mathsf{true}\} \\
&\mathsf{neededIDs}(/\mathsf{atmdCall} \ x \ y/(T_1 \ \ldots \ T_n)) = \{x\} \ \cup \mathsf{neededIDs}(T_1) \ \cup \cdots \cup \mathsf{neededIDs}(T_n) \\
&\mathsf{neededIDs}(/x/\mathsf{comb} \ n \ s' \ /i/E(s \ldots) \ T) = (\mathsf{neededIDs}(/i/E) \ \cup \mathsf{neededIDs}(T)) \ - \{x\} \\
&\mathsf{neededIDs}(n\mathsf{o}) = \quad \emptyset \quad \mathsf{neededIDs}(/x/(s \leftarrow T) \ \ldots \ |s' \leftarrow /i'/E', /i/E) = \mathsf{neededIDs}(/i/E) \ \cup \\
&\mathsf{neededIDs}(/i'/E') \quad \mathsf{neededIDs}(/x/(s \leftarrow T) \ \ldots \ |, /i/E) = \mathsf{neededIDs}(/i/E)
\end{aligned}$$

图 15. 进展所需关系

$$\begin{aligned}
&\text{大写标识符}(n) = \quad \emptyset \quad \text{大写标识符}(/x/s) = \quad \emptyset \quad \text{大写标识符}(/\text{值}/(T_1 \ \ldots \ T_n)) = \text{大写标识符}(T_1) \ \cup \cdot \cdot \ \cup \\
&\text{大写标识符}(T_n) \quad \text{大写标识符}(/\mathsf{freshCall}/(T_1 \ \ldots \ T_n)) = \text{大写标识符}(T_1) \ \cup \cdot \cdot \ \cup \text{大写标识符}(T_n) \quad \text{大写标识符}(/\mathsf{atmdCall} \ x \ y/(T_1 \ \ldots \ T_n)) = \text{大写标识符}(T_1) \ \cup \cdot \cdot \ \cup \text{大写标识符}(T_n) \quad \text{大写标识符}(/x/\mathsf{comb} \ n \ s' \ /i/E(s \ldots) \ T) = \quad \emptyset \quad \text{大写标识符}(n\mathsf{o}) = \quad \emptyset \quad \text{大写标识符}(/x/(s \leftarrow T) \ \ldots \ |s' \leftarrow /i'/E', /i/E) = \text{大写标识符}(/i/E) \\
&\cup \text{大写标识符}(/i'/E') \ \text{若需要标识符}(/x/(s \leftarrow T) \ \ldots \ |/i'/E', /i/E) \ = \emptyset \quad \text{大写标识符}(/x/(s \leftarrow T) \ \ldots \ |, /i/E) = \\
&\text{大写标识符}(/i/E) \ \text{若需要标识符}(/x/(s \leftarrow T) \ \ldots \ |, /i/E) \ = \emptyset \quad \text{大写标识符}(/x/(s \leftarrow T) \ \ldots \ |s' \leftarrow /i'/E', /i/E) \\
&= \{x\} \ \cup \text{大写标识符}(/i/E) \ \cup \text{大写标识符}(/i'/E') \ \text{若需要标识符}(/x/(s \leftarrow T) \ \ldots \ |/i'/E', /i/E) \ \neq \emptyset \quad \text{大写标识符}(/x/(s \leftarrow T) \ \ldots \ |, /i/E) = \{x\} \ \cup \text{大写标识符}(/i/E) \ \text{若需要标识符}(/x/(s \leftarrow T) \ \ldots \ |, /i/E) \ \neq \emptyset
\end{aligned}$$

图 16. 进展所需上界关系

恢复形式(*n*) = ∅恢复形式(/*x*/*s*) = ∅恢复形式(/值/(*T*1 ... *Tn*)) = 恢复形式(*T*1) ∪ · ·· ∪恢复形式(*Tn*) 恢复形式(/freshCall/(*T*1 ... *Tn*)) = 恢复形式(*T*1) ∪ · ·· ∪恢复形式(*Tn*) 恢复形式(/ atmdCall *x y*/(*T*1 ... *Tn*)) = {*y*} ∪恢复形式(*T*1) ∪ . . ∪恢复形式(*Tn*) 恢复形式(/*x*/comb *n s*′ /*i*/*E*(*s* . . .) *T*) = ∅恢复形式(*n*o) = ∅恢复形式(/*x*/(*s* ← *T*) ... |*s*′ ← /*i*′/*E*′, /*i*/*E*) = ∅恢复形式(/*x*/(*s* ← *T*) ... |, /*i*/*E*) = ∅

图 17. 进展所需的无限恢复关系

$$lookup(s, (s_1 \leftarrow V_1)(s_2 \leftarrow V_2) \ \ldots \ |s' \leftarrow /i_x/E, /i'_{x'}/E') = \quad lookup(s, (s_2 \leftarrow V_2) \ \ldots \ |s' \leftarrow /i_x/E, /i'_{x'}/E')$$
$$lookup(s, (s \leftarrow V) \ \ldots \ |s' \leftarrow /i_x/E, /i'_{x'}/E') = \quad Vlookup(s, \ |s \leftarrow /i_x/E, /i'_{x'}/E') = /i_x/Elookup(s, \ |s' \leftarrow /i_x/E, /i'_{x'}/E') = \quad lookup(s, E')$$

图 18. 查找

$$\begin{aligned}
returnOk(n, i) &= \text{true} \\
returnOk(o, i) &= \text{true} \\
returnOk(/\emptyset/s, i) &= \text{true} \\
returnOk(/\text{val}/(V \ldots), i) &= \text{true} \\
returnOk(/i'/C, i) &= \neg IDin(i, /i'/C) \\
returnOk(E, i) &= \neg IDin(i, E) \\
\\
returnOk(/y/(\langle -1 \ \textbf{veval} \rangle \ z \ /i'_{x'}/E), i) &= returnOk(/i'_{x'}/E, i) \\
returnOk(/y/(f \ V \ldots), i) &= \neg takesDE(f) \wedge \bigwedge returnOk(V, i) \ldots
\end{aligned}$$

图 19. returnOk

IDin（图 20）非常简单，但将其单独提取出来，可以让 returnOk 更易读：

$$IDin(x, i) \quad = \quad i \in \text{neededIDs}(x) \vee i \in \text{upperIDs}(x)$$

图 20. IDin

“takesDE”（参见 21）同样很简单：如果原始组合器或派生组合器接收其动态环境，则返回 true。

$takesDE(0\ \mathbf{vau}) = 真$

$takesDE(0\ \mathbf{零判断}) = 真 takesDE$

$(-1\ \mathbf{vif0}) = 真 takesDE(x\ \mathbf{y}) = 假$

（其他原语不会）

$takesDE(/i''/\mathbf{comb}\ n\ s'/i'_r/E'(s\ldots)\ Tb) = \quad s' \neq 0 takesDE(x) = \quad 真$

（为确保安全，其他任何项，包括挂起项，都计为 true）

图 21. takesDE

dropRV 处理两种主要情况——第一种是对 veval 的调用，如果它是多余的（显式环境的标识符与当前动态环境的标识符相同），就会将其移除。第二种是挂起函数调用，此时会对所有参数递归调用 dropRV。如果这确实改变了该调用，我们就会再次对其进行部分求值，因为参数可能因此变得更简单，从而可以进一步求值。

$dropRV(/y/(-1\ \text{veval}\ z/i_f/E), /i_x/E, ES, FS) = \quad dropRV(z, /i_x/E, ES, FS)$（仅当 $y \neq$ 值时）$dropRV(/y/(n\ \text{o}\ V\ldots), /i_x/E, ES, FS) = \text{let}\ g = (n\ \text{o}\ V\ldots)\ \text{let}\ z = (n\ \text{o}\ dropRV(V, /i_x/E, ES, FS)\ldots)\ \text{in}[\text{peval}\ /\text{freshCall}/z, /i_x/E/i_x/E\ ES\ FS]\ \text{if}\ z \neq g \text{else}\ /y/z$（仅当 $y \neq$ 值 $n \neq -1$时）$dropRV(\ /y/(/i''/\text{comb}\ n\ s'/i'_r/E'(s\ldots)\ Tb\ V\ldots), /i_x/E, ES, FS) = \text{let}\ g = (/i''/\text{comb}\ n\ s'/i'_r/E'(s\ldots)\ Tb\ V\ldots)\ \text{in let}\ z = (/i''/\text{comb}\ n\ s'/i'_r/E'(s\ldots)\ Tb dropRV(V, /i_x/E, ES, FS)\ldots)\ \text{in}[\text{peval}\ /\text{freshCall}/z, /i_x/E/i_x/E\ ES\ FS]\ \text{if}\ z \neq g \text{else}\ /y/z dropRV(z, /i_x/E, ES, FS) = \quad z$否则，原样返回

图 22. dropRV

[combine 0 eval (V_1 V_2) /i_x/E ES FS] → [combine /i''/−1 veval (unval(V_1) V_2) /i_x/E ES FS][combine −1 veval (V/$i'_{x'}$/E') /i_x/E ES FS] → let V' = [peval V/$i'_{x'}$/E' (/$i'_{x'}$/E' ∪ ES) FS] in[在 V' 下 (−1 veval V' /$i'_{x'}$/E') /i_x/E ES FS][在 z 下 (−1 veval V' /$i'_{x'}$/E') /i_x/E ES FS] → let o = $returnOk$(z, i') in$dropRV$(z, /i_x/E, ES) if o = true else /atmdCall ∅ ∅/(−1 veval V' /$i'_{x'}$/E')[combine 0 vau (s' (s . . .) V) /i_x/E ES FS] → [peval /$genID$()/comb 0 s' /i_f/E(s . . .) $unval$(V) /i_x/E ES FS][combine 0 包装 (/i''/comb n s' /i'/E' (s . . .) V) /i/E ES FS] → /i''/comb (S n) s' /i'/E' (s . . .) V[combine 0 解封装 (/i''/comb (S n) s' /i'/E' (s . . .) V) /i/E ES FS] → /i''/comb n s' /i'/E' (s . . .) V[combine 0 零判断 (V_c V_t V_e) /i_x/E ES FS] → let F = (0 零判断 V_c V_t V_e)in let FS' = {F} ∪ FSin let V'_c = [peval unval(V_c) /i_x/E ES FS] in let V'_t = unval(V_t) if F ∈ FSelse [peval unval(V_t) /i_x/E ES FS'] in let V'_e = unval(V_e) if F ∈ FSelse [peval unval(V_e) /i_x/E ES FS'] in[在 V'_c 下(−1 vif0 V'_c V'_t V'_e) /i_x/E ES FS][在 0 下 (−1 vif0 V' V_t V_e) /i_x/E ES FS] → [peval V'_t/i_x/E ES FS][在 (S n) 下 (−1 vif0 V'_c V'_t V'_e) /i_x/E ES FS] → [peval V'_e/i_x/E ES FS][在 z 下 (−1 vif0 V'_c V'_t V'_e) /i_x/E ES FS] → /atmdCall ∅ ∅/(−1 vif0 z V'_t V'_e)（if 的其他情况）[combine 0 整数转符号 (n) E ES FS] → /∅/$'sn$（由数字 n 构成的符号）[combine 0 数组 (V . . .) E ES FS] → /值/(V . . .)

图 23. 部分求值原语的语义

B 部分求值伪代码

算法 1： 部分求值算法伪代码（不包括调用）

```
定义 部分求值(形式, 环境, 环境_栈, 正在求值的_形式_集合)： /*                                    */
    数据： 我们的输入是当前要进行部分求值的形式,
    用于对其进行求值的环境 ,
    当前活动环境的完整调用栈,
    以及当前正在进行的求值集合。
    结果： 部分求值后的形式
    /* 首先，我们检查部分求值此形式是否会取得任何进展 */
    not_yet_evaled ← true ∈? needed标识符(x);
    newly_real_env_ids ← ∃ i ∈ neededIDs(x) 使得 /i_x/E ∈ env_stack;
    newly_unblocked ← evaluating_forms_set ∩ 恢复形式(x) ≠ 恢复形式(x);
    如果 not_yet_evaled ∨ newly_real_env_ids ∨ newly_unblocked：
        如果 形式是一个环境值，/i_x/E：
            /* 获取此环境较新的真实版本，                    如果存在                    */
            若 /i_r/E′ ∈ env_stack：
                返回 /i_r/E′
            否则：
                返回 /i_x/E;
        否则如果 形式是一个派生组合器值，/i/⟨comb n s /i′_x/E(s . . . ) Tb⟩：
            /* 对组合器的静态环境和函数体进行递归                                          */
            如果 x =? r：
                返回 /i/comb n s /i′_x/E(s . . . ) Tb;
            否则：
                new_env ← (s ← /i/s) . . . |s′ ← /i/s′, env;
                new_body ←
                  部分求值(Tb, new_env, new_env ∪ env_stack, evaluating_forms_set);
                返回 comb n s env(s . . . ) new_body;
        否则：
            /* 这是一个调用——在算法 2 中单独列出                                           */
            返回 PartialEvalCall(形式, 环境, 环境_栈, 正在求值的_形式_集合);
    否则：
        /* 部分求值不会取得任何进展，直接原样返回形式                                        */
        返回形式;
```

算法 2： 部分求值算法伪代码（用于调用）

定义 PartialEvalCall(形式，环境，环境_栈，求值中_形式_集合)： /* */
    数据： 我们的输入是当前要进行部分求值的形式，
    用于对其求值的环境 ，
    当前活动环境的完整调用栈，
    以及当前正在进行的求值集合。
    结果： 部分求值后的形式
    /* 首先对组合器进行部分求值 */
    $/x/(T_1\ T_2 \ldots) \leftarrow form$;
    $c \leftarrow$ 部分求值$(T_1, env, env_stack, evaluating_forms_set)$;
    /* 如果结果是挂起的符号查找或挂起调用，则返回，因为我们无法
       取得进展 */
    如果 $c = /x/s \vee c = /\text{atmdCall}\ x\ y/(T' \ldots)$**：**
        返回 /atmdCall $\emptyset\ \emptyset/(c\ T_2 \ldots)$;
    /* 否则 ，c 是组合器 ，原始组合器或派生组合器。获取其包装层级 */
    if $c = \langle n\ o \rangle$**：**
        $wrap\_level \leftarrow n$
    否则：
        $/i/\textbf{comb}\ n\ s'\ /i'_x/E'\ (s \ldots)\ Tb \leftarrow c$
        $wrap\_level \leftarrow n$
    /* 对参数求值，直到包装_层级为 0（如果层级为 -1，则完全不求值） */
    $args \leftarrow (T_2 \ldots)$
    如果 $wrap\_level > 0$**：**
        $args \leftarrow$(对 $T \in args$ 执行部分求值$(T, env, env_stack, evaluating_forms_set)$);
        如果 *args* 中的任意条目不是值**：**
            返回 /atmdCall $\emptyset\ \emptyset/(c\ args \ldots)$;
        重复
            对 $T \in args$ 执行 $args \leftarrow (unval(T))$;
            $args \leftarrow$（对 $T \in args$ 执行部分求值$(T,\ env,\ env_stack,\ evaluating_forms_set)$);
            如果 *args* 中有任何条目不是值**：**
                $new\_c = replacewraplevel(c,\ wrap\_level)$
                返回 /atmdCall $\emptyset\ \emptyset/(new\_c\ args \ldots)$;
        直到 $wrap\_level = 0$;
    if $c = \langle n\ o \rangle$**：**
        return $drop\_redundant\_eval(o(env, env\_stack, args)\ env, env\_stack)$;
    否则：
        /* 创建我们的内部环境 */
        $/i/\textbf{comb}\ n\ s'\ /i'_x/E'\ (s \ldots)\ Tb \leftarrow c$
        $inner\_env \leftarrow /i/(s \leftarrow V$**适用于** $(s,\ V) \in zip((s \ldots)\ args)\ |s' \leftarrow env,\ /i'_{x'}/E'$;
        /* 检查我们是否已经在求值此形式， 以防止无限递归 */
        $new\_c = replaceWrapLevel(c,\ wrap\_level)$
        如果 $(Tb, inner\_env) \in evaluating\_forms\_set$**：**
            返回 /atmdCall $\emptyset(Tb, inner\_env)/(new\_c\ args \ldots)$;
        $result \leftarrow$
          PartialEval$(Tb,\ inner_env,\ \{inner_env\} \cup env_stack,\ \{(Tb,\ inner_env)\} \cup evaluating_forms_set)$;
        如果组合器_返回_*ok(result, env.id)*：
            返回 drop_冗余_eval(result, env, env_stack);
        elif $s' \neq \emptyset$**：**
            /* 如果此组合器接收动态环境， 跟踪当前动态环境
               此调用继续进行所需的环境标识符 */
            返回 /atmdCall $env.id\ \emptyset/(new\_c\ args \ldots)$;
        否则：
            返回 /atmdCall $\emptyset\ \emptyset/(new\_c\ args \ldots)$;