Common Lisp 中指数级宏膨胀的重构札记，或：多个 @Body 被视为有害

Michael Wessel
michael.wessel@sri.com
SRI International 门洛帕克，
加利福尼亚州，美国

arXiv:2305.02991v1 [cs.PL] 4 May 2023

摘要

我最近受聘为一个规模庞大的 Common Lisp 项目提供咨询，该项目的代码（包括注释）超过一百万行。下文称之为“System X”。System X 遭遇了编译时间过长的问题：使用 ACL 10.1，在配备固态硬盘和 16 GB 内存、搭载 3.1 GHz 英特尔酷睿 i7 的 MacBook Pro 上，完整重新编译耗时约 33 分 17 秒。后来发现，若干宏导致了指数级代码膨胀。对这些宏进行重构后，系统重新编译仅需 5 分 30 秒，速度提升了 ≈ 6 倍。在这篇经验报告中，我将先阐明问题，再介绍两种运用宏重构技术的潜在解决方案。这些技术也适用于相关场景。

关键词

Common Lisp、宏、指数级代码膨胀、宏重构、超大型 Lisp 系统

1 引言

宏，以及在语言本身中编写语言扩展程序的能力，是 Lisp 家族许多成员最受欢迎且最强大的特性之一；Common Lisp 尤其如此[8]，，John Foderaro 曾将其称为可编程的编程语言。在宏展开／编译时能够使用完整的编程语言，使 Common Lisp 成为实现领域专用语言的理想平台[6, 9]，，而且一直如此（例如，Lisp 早期就曾是面向对象编程概念的目标平台 [3]）。与大多数其他编程语言中的宏不同，Common Lisp 允许用同一种语言定义宏。得益于其同像性，它提供了统一的“程序即数据”表示方式，并允许在语言本身中构造、操作，尤其是计算宏展开结果。语言的全部能力始终可用——不仅在运行时如此，在宏展开（“编译”）时也是如此 [4, 5, 7]。

一如既往，能力越大，责任越大：宏可能是一把双刃剑。在 Common Lisp 这类语言中，这一点尤其明显，因为其主要开发模式并非传统的“编辑 — 完整重新编译 — 调试”循环，而是基于增量式重新定义、求值和编译的交互式动态模式。对代码库所做更改（尤其是宏更改）带来的意外后果，有时可能会长时间未被察觉，特别是在系统完整重新编译被推迟的情况下。对于规模较大的项目和团队，这一点尤为突出。一旦编译时间超过半小时，开发人员在日常开发中就会避免完整重新编译，而构建系统则会通常会受托在夜间生成新的基础映像，其中包含多位开发人员所做的更改。当然，回归问题会每天记录并监控。但即使构建系统报告了构建时间和快速加载（*FASL*）文件的大小，也未必能清楚地判断究竟是哪些更改延长了构建时间——毕竟，构建系统可能只是那晚运行不顺，还忙着执行备份，诸如此类。

因此，要将系统中意料之外的行为追溯到代码库中（已不算近期的）改动，可能会变得更加困难。正因如此，Common Lisp 代码的增量编译可能反而成了缺点。我建议开发人员不仅要检查代码改动是否导致系统的语义和功能特性发生意外变化，还要检查非功能特性（例如 FASL 文件大小和构建时间）。尤其要关注宏。

我最近有机会参与了“System X”的工作。这是一个规模极其庞大、历经数十年打造的 Common Lisp 系统，代码超过一百万行。System X 饱受编译时间过长之苦：在使用 ACL 10.1 的 3.1 GHz MacBook Pro（英特尔酷睿 i7）、SSD 和 16 GB 内存上，完整编译需要 33:17 分钟。结果发现，三个设计糟糕的宏 导致了指数级代码膨胀。重构这些宏后，完整重新编译只需 5:30 分钟，速度提升了 ≈ 6倍。

在这份经验报告中，我将首先阐明问题，然后通过宏重构展示两种可能的解决方案。重构方法的有效性不仅体现在编译时间缩短了 6 倍，还体现在 FASL 文件大小的减少上。所采用的方法论可以应用于相关场景。最后，我将提出一些建议。

2 问题

System X的问题可以通过几个合成示例轻松说明。请看图1中的建立上下文的 with-bad 宏。与许多 with- 宏一样，它利用具有动态作用域的特殊变量 （此处为 *a*）来控制其词法作用域之外的运行时行为。在该示例中，动态变量 *a* 的绑定决定了 with-bad 宏的展开是打印 x 还是 (1+ x)。

这个宏用于说明指数级宏展开的问题。with-宏经常会相互嵌套，这可能会让开发人员难以察觉这类问题的严重程度。例如，该宏可能是网站 HTML 生成框架的一部分，因此包含 with-head、with-body、with-table 等宏。复杂的网页不仅会包含许多层层嵌套的这类宏，而且这类网页中的某些常见设计元素也可能会...

```lisp
(defvar *a* nil)

(defmacro with-bad ((x) &body body)
  `(if *a*
       (progn
         (princ ,x)
         ,@body)
       (progn
         (princ (1+ ,x))
         ,@body)))

(defmacro blowup ()
  `(with-bad (3)
     (with-bad (4)
       (with-bad (5)
         (with-bad (6)
           'done)))))
```

```lisp
(if *a*
    (progn (princ 3)
      (if *a*
          (progn (princ 4)
            (if *a*
                (progn (princ 5) (if *a* (progn ... 'done)))
                (progn (princ (1+ 5)) (if *a* (progn ... 'done)))))
          (progn (princ (1+ 4))
            (if *a*
                (progn (princ 5) (if *a* (progn ... 'done)))
                (progn (princ (1+ 5)) (if *a* (progn ... 'done))))))))
  (progn (princ (1+ 3))
    (if *a*
        (progn (princ 4)
          (if *a*
              (progn (princ 5) (if *a* (progn ... 'done)))
              (progn (princ (1+ 5)) (if *a* (progn ... 'done)))))
        (progn (princ (1+ 4))
          (if *a*
              (progn (princ 5) (if *a* (progn ... 'done)))
              (progn (princ (1+ 5)) (if *a* (progn ... 'done))))))))
```

图1：具有指数级宏展开的宏

```lisp
(defmacro with-good ((x) &body body)
  (let ((fn (gensym)))
    `(flet ((,fn () ,@body))
       (if *a*
           (progn
             (princ ,x)
             (,fn))
           (progn
             (princ (1+ ,x))
             (,fn))))))

(defmacro no-blowup ()
  `(with-good (3)
     (with-good (4)
       (with-good (5)
         (with-good (6)
           'done)))))
```

```lisp
(flet ((#:g1381515 nil
        (flet ((#:g1381516 nil
                (flet ((#:g1381517 nil
                        (flet ((#:g1381518 nil 'done))
                          (if *a*
                              (progn (princ 6) (#:g1381518))
                              (progn (princ (1+ 6)) (#:g1381518))))))
                  (if *a*
                      (progn (princ 5) (#:g1381517))
                      (progn (princ (1+ 5)) (#:g1381517))))))
          (if *a*
              (progn (princ 4) (#:g1381516))
              (progn (princ (1+ 4)) (#:g1381516))))))
  (if *a*
      (progn (princ 3) (#:g1381515))
      (progn (princ (1+ 3)) (#:g1381515))))
```

图 2：具有线性宏展开的宏

（常见的页眉、页脚和导航菜单）已被聚合为更高层次的宏，这些宏又被用于其他宏中，依此类推。

需要注意的是，Common Lisp 并不包含递归宏展开功能 macroexpand-all；它只提供 macroexpand-1。这只能执行单层宏展开。不过，也有第三方解决方案可供使用。我使用了其中一个软件包来诊断 System X [1, 2] 中的问题。

考虑图 1 中包含四个嵌套 with-bad 出现的 blowup 的宏展开，我们可以清楚地看到，由于重复的 ,@body 形式，其大小相对于原始定义呈指数增长。一般而言，给定为 n的嵌套深度，展开后的宏代码大小为 2^n。我甚至在实际代码中发现了包含两个以上 ,@body 形式的 with- 宏；一般而言，包含 m,@body 形式的 with- 宏在嵌套 n次时会展开为大小为 m^n 的代码。这显然应该避免。

有时，这种指数级宏很容易修复。对于图 1，只需将 ,@body 形式移入局部函数定义（flet），并在运行时的两个位置调用该局部函数，而不是重复代码。由此得到的图 2 所示的 with-good 宏，其展开大小现在随嵌套深度呈线性增长，而非指数级增长。

遗憾的是，指数级宏并非总能轻易修复。例如，假设 with-good 中的宏参数 x 用于为 ,@body 内部的代码建立词法绑定，而不只是作为宏的“输入参数”。在这种情况下，宏内部会使用 (let ((,x ...)) ... ,@body) 为 ,x 建立相应的词法作用域。此外，,x 的具体绑定可能取决于复杂的运行时和编译时条件。特别是，,x 的值可能取决于 *a* 的运行时值，而该值在编译时／宏展开时是未知的，因此无法通过代码重写／转换预先确定。因此，正确建立词法上下文至关重要，例如通过局部函数的形参列表来建立。

我在 System X 中需要重构的指数宏就属于这种“更难”的类型。与其揭示这个宏的细节，我将在下文中使用图 3 中的合成示例。这个宏的复杂程度与之相似，可用于说明其中的问题及可能的解决方案。

```lisp
(defvar *within-recording* nil)
(defvar *recording-stack* nil)

(defmacro with-bad-recording ((nested-p) &body body)
  `(if *within-recording*
       (let ((,nested-p t))
         (declare (ignorable ,nested-p))
         ,@body)
     (let* ((*recording-stack* nil)
            (*within-recording* t)
            (,nested-p nil))
       (declare (ignorable ,nested-p))
       ,@body
       (process-recordings))))

(defun process-recordings ()
  (pprint *recording-stack*))

(defun do-something-and-record (x)
  (push x *recording-stack*))

(defmacro test-bad-recording ()
  `(with-bad-recording (n-p)
     (if n-p
         (do-something-and-record 11)
       (do-something-and-record 1))
     (with-bad-recording (n-p)
       (do-something-and-record 2)
       (with-bad-recording (n-p)
       (if n-p
           (do-something-and-record 3)
         (do-something-and-record 33))))))

(test-bad-recording)
;; -> (3 2 1)
```

图 3：System X 中发现的指数级 `with-`宏反模式

with-bad-recording 背后的理念是建立一个动态作用域的上下文，用于跟踪被记录到栈中的“指令”；这些指令可以是日志文件条目、用于输出记录演示历史的内容等。隐藏的特殊变量 *recording-stack*（具有动态作用域）用于跟踪栈中的值。这个特殊变量不应对用户代码可见（它属于“内部”变量）；相反，应使用诸如 (do-something-and-record x) 这样的访问器函数（或宏）来操作它。

此外，不知为何，with-bad-recording 的调用方也想知道当前调用是在顶层进行，还是在运行时已经属于一次“嵌套”调用；因此可以传入变量 nested-p，随后分别将其绑定为 nil 或 t。为了判断这一点，宏的原作者还引入了另一个特殊变量 *within-recording*（或许有些多余）。同样，这是一个不应让用户代码看到的“内部”特殊变量，因此才提供了 nested-p。test-bad-recording 展示了一个用例。请注意，n-p（即宏中的 nested-p）的运行时值无法从词法定义中得知，因为 test-bad-recording 在运行时可能嵌套于另一个 with-bad-recording 上下文中。该用例显示，最终栈中保存的是 (3 2 1)。

```lisp
(defmacro with-recording-v1 ((nested-p) &body body)
  (let ((fn (gensym)))
    `(flet ((,fn () ,@body))
       (if *within-recording*
           (let ((,nested-p t))
             (declare (ignorable ,nested-p))
             (,fn))
         (let* ((*recording-stack* nil)
                (*within-recording* t)
                (,nested-p nil))
           (declare (ignorable ,nested-p))
           (,fn)
           (process-recordings))))))

(defmacro test-recording-v1 ()
  `(with-recording-v1 (n-p)
     (if n-p
         (do-something-and-record 11)
       (do-something-and-record 1))
     (with-recording-v1 (n-p)
       (do-something-and-record 2)
       (with-recording-v1 (n-p)
         (if n-p
             (do-something-and-record 3)
           (do-something-and-record 33))))))

(test-recording-v1)
;; **++++ Error in (TOP-LEVEL-FORM 14):
;;   The variable N-P is unbound.
```

图 4：将多个 ,@body 替换为局部函数调用，导致宏失效——请参见编译器警告

显然，这个宏现在可能导致指数级宏展开，而 test-bad-recording 已经受到这种 blowup 的影响。我们能否像图 2 中那样修复这个宏?

3 第一种解决方案——使用FLET进行重构

第一种解决方案是对我们在第 2 节中讨论过的思路进行推广——我们将每个 ,@body 替换为对一个局部函数的调用，该函数包含单个 ,@body。为此可以使用 FLET 或 LABELS。

图 4 展示了修复 with-bad-recording 的一种朴素尝试。现在这个宏显然已经失效，因为 ,@body 引用了 ,nested-p = n-p，而它在外层 flet 中不可见，因此编译器警告该变量现在未绑定。显而易见的解决方案是将 ,nested-p = n-p 设为局部函数的参数，这样 ,@body 所需的词法变量便由局部函数建立。图 5 展示了这一做法。

如果某个分支中修改了词法变量，情况就会更加复杂——请看图 6 中展示的 with-bad-recording-v2 变体，其中 nested-p 被 control-p 替换。control-p 的值会影响输出，而且它既可能在用户提供的 ,@body 代码中设置，也可能在宏本身内部设置。重构这样的宏就不再那么机械了，需要格外谨慎，以确保建立正确的词法环境。

```lisp
(defmacro with-recording-v2 ((nested-p) &body body)
  (let ((fn (gensym)))
    `(flet ((,fn (,nested-p)
              (declare (ignorable ,nested-p))
              ,@body))
       (if *within-recording*
           (,fn t)
         (let* ((*recording-stack* nil)
                (*within-recording* t))
           (,fn nil)
           (process-recordings))))))

(defmacro test-recording-v2 ()
  `(with-recording-v2 (n-p)
     (if n-p
         (do-something-and-record 11)
       (do-something-and-record 1))
     (with-recording-v2 (n-p)
       (do-something-and-record 2)
       (with-recording-v2 (n-p)
         (if n-p
             (do-something-and-record 3)
           (do-something-and-record 33))))))

(test-recording-v2)
```

图 5：使用局部函数变量建立词法上下文——请注意，动态上下文仍由原始分支结构建立

```lisp
(defun process-recordings-v2 (control-p)
  (if control-p
      (pprint *recording-stack*)
    (pprint :foobar)))

(defmacro with-bad-recording-v2 ((control-p) &body body)
  `(if *within-recording*
       (let ((,control-p t))
         (declare (ignorable ,control-p))
         ,@body)
     (let* ((*recording-stack* nil)
            (*within-recording* t)
            (,control-p nil))
       (declare (ignorable ,control-p))
       ,@body
       (process-recordings-v2 ,control-p))))

(defmacro test-bad-recording-v2 ()
  `(with-bad-recording-v2 (control-p)
     (do-something-and-record 1)
     (with-bad-recording-v2 (control-p)
       (do-something-and-record 2)
       (with-bad-recording-v2 (control-p)
         (do-something-and-record 3)))
     (setf control-p nil))) ; nil -> :foobar, t -> (3 2 1)

(test-bad-recording-v2)
```

图 6：如果作为宏参数的词法变量会被修改，例如 `control-p`，那么重构就会更复杂

特别是，我们意识到，必须在局部函数内部 更改 control-p 的值，这样调用

```lisp
(defmacro with-recording-v3 ((control-p) &body body)
  (let ((fn (gensym)))
    `(flet ((,fn (,control-p branch-p)
              ,@body
              (when branch-p
                (process-recordings-v2 ,control-p))))
       (if *within-recording*
           (,fn t nil)
         (let* ((*recording-stack* nil)
                (*within-recording* t))
           (,fn nil t))))))

(defmacro test-recording-v3 ()
 `(with-recording-v3 (control-p)
     (do-something-and-record 1)
     (with-recording-v3 (control-p)
       (do-something-and-record 2)
       (with-recording-v3 (control-p)
         (do-something-and-record 3)))
     (setf control-p nil))) ; nil -> :foobar, t -> (3 2 1)
```

图 7：通过 `branch-p` 参数，在一个函数中容纳不同的宏分支

(process-recordings-v2 ,control-p) 将从正确的词法作用域中获取 ,control-p 的值。图 6 展示了一种可能的解决方案；branch-p 参数用于告知局部函数调用上下文，而局部函数则必须根据该参数的值，容纳或“模拟”宏原始分支中不同的运行时行为。这并非总是可行，但在本例中相当简单。[1]

至此，第一种重构策略的讨论告一段落。简而言之，保留原始分支结构，以建立不同的词法作用域和动态作用域。必须找到一种通用的 ,@body 形式，使其能够复现原有的运行时行为，并将其置于局部函数中。局部函数负责建立所需的词法上下文，并从原始分支结构中移除这些上下文。使用 branch-p 等额外的控制参数来选择特定于分支的运行时行为。具体来说，我们保留原始宏的分支结构，以便为特殊变量建立正确的绑定，并通过相应地调用局部函数来设置正确的词法上下文。

刚才讨论的重构模式的一个潜在缺点是引入了额外的局部函数，以及额外函数调用所带来的运行时开销。[2]

更严重的是，重构后的宏现在会掩盖宏的原始结构——它变成了“里外颠倒”，因为我们采用了函数组合 来实现程序顺序执行； 如图 2 所示，(princ 6) 现在在文本顺序上位于 (princ 3) 之前，这与原始定义相反。具体来说，局部函数中的 ,@body 现在与原始分支结构“脱离”了，使得宏展开更难理解。

如果这些问题确实值得重视，则可以采用以下替代性重构策略。

[1]请注意，目前我们暂时忽略这些宏的返回值可能存在的差异；通常，with- 宏不会返回值，但这只是一种约定，并非严格要求。[2]不过，也可以将这些局部函数声明为内联函数。

4 第二种解决方案——使用 PROGV 进行重构

在下文中，我们并不 使用一个可从重写后的宏的不同分支中调用的局部函数。相反，我们试图将原始的分支结构统一为一个 共同结构，而原始结构会建立不同的词法上下文和动态上下文。事实证明，为特殊变量建立正确的（条件性）绑定是最大的障碍，为此我们将使用 progv。

以下步骤可以理解为一种保留语义的代码重写过程／转换器。我们将对 with-bad-recording 宏应用以下步骤，以驯服指数级增长的怪兽，并将其重写为线性宏：

步骤 1 对所有分支特殊形式（unless、when、cond、.....）进行宏展开／重写，将其转换为 if（在我们的示例中，已经是这种情况，因此此步骤不适用）：(if <condition>(let (<binding 11>... <binding 1n>) ,@body) (let (<binding 21> ... <binding 2m>) ,@body))

步骤 2 确保所有分支中的所有 let 都引用相同的变量，且顺序一致。如果 <binding ij> = (,var val) 中的 var 是宏参数，那么所有分支中都必须已经包含有效的 (,var val) 绑定。否则，在（某些分支的）,@body 中，var 就会未绑定（例如，该宏从一开始就有缺陷）。

如果 var 是一个特殊变量，即 *var*，那么，当该分支未包含 <绑定> = (*var* val) 时，我们暂时引入一个“虚拟”绑定 <binding> = (*var* *var*)。其目的是表达我们打算不动态改变 *var* 的绑定。请注意，当 *var* 被用作 “ 只读 ” 变量时，这没有问题；但在 with-bad-recording 等情况下则会有问题，因为其中修改了 *recording-stack*；解决方案见下文。

因此，现在每个 let 中都有相同数量的 (var val) 绑定，其中的 val 可能（但不一定）不同；注意，$max(n, m) \le k \le n+m$:

```
(if <condition>
     (let ( (<var1 val11>) ... (<vark val1k>) )
          ,@body)
  (let ( (<var1 val21>)  ... (<vark val2k>) )
       ,@body))
```

步骤 3 接下来，我们移除不同的分支，在单个 let 中建立所有绑定，并通过在 let 绑定形式内部建立不同的绑定来恢复<条件> 的效果。由于我们通过将条件移入 let 的 lambda 列表中进行提取，移除了周围代码中的不同分支，因此也消除了多个 ,@body 的出现：

```
(let ( ( <var1> (if <condition> <val11> <val21>) )
         ...
       ( <vark> (if <condition> <val1k> <val2k>) ))
  ,@body)
```

```
(defmacro with-recording-v4 ((nested-p) &body body)
  (let ((ret-val (gensym))
        (old-within-recording (gensym)))
    `(let* ((*recording-stack*
             (if *within-recording*
                 *recording-stack*
               nil))
            (,old-within-recording *within-recording*)
            (,nested-p *within-recording*)
            (*within-recording* t))
       (declare (ignorable ,nested-p))
       (progn
         (setf ,ret-val
           (multiple-value-list
            (progn
              ,@body)))
         (if ,old-within-recording
             (values ,ret-val)
           (process-recordings))))))

(defmacro test-recording-v4 ()
  `(with-recording-v4 (n-p)
     (if n-p
         (do-something-and-record 11)
       (do-something-and-record 1))
     (with-recording-v4 (n-p)
       (do-something-and-record 2)
       (with-recording-v4 (n-p)
       (if n-p
           (do-something-and-record 3)
         (do-something-and-record 33))))))

(test-recording-v4)
;; -> (1)
```

图 8：遮蔽*recording-stack* 破坏了宏

步骤 4 到目前为止都很顺利——不过有一个问题：这只适用于以“只读”方式使用的动态作用域变量。正如前面提到的，我们引入了一个“虚设”绑定<绑定> =(*recording-stack* *recording-stack*)，以表示我们希望保持 *recording-stack* 的绑定不变。 但我们通过建立一个新的绑定框架改变了它——我们“遮蔽”了先前建立的绑定。使用 let/let*，对此无计可施。

其效果如图 8 所示——现在，重构后的宏显然已经出错，示例调用 (test-recording-v4) 展示了这一点。原本应像原始版本一样返回 (3 2 1)，但现在我们只能得到压入栈中的第一个值：(1)。

不过，这些代码重写让我们走上了正确的道路。我们只需避免在不希望更改动态变量的情况下进行遮蔽。幸运的是，Common Lisp 提供了相应的解决方案；我们应该称赞 Common Lisp 的设计者预见到了这种情况：progv 可以这样完成这项工作：

```
(progv (when 更改-
*var* (list '*var*))
(list val)
```

如果 alters-*var* = T，该形式等价于 (progv '(*var*) (list val))，因此建立了一个新的绑定

```lisp
(defmacro with-recording ((nested-p) &body body)
  (let ((ret-val (gensym))
        (old-within-recording (gensym)))
    `(let* ((,old-within-recording *within-recording*)
            (,nested-p *within-recording*)
            (*within-recording* t))
       (declare (ignorable ,nested-p))
       (progv
           (unless ,old-within-recording
             (list '*recording-stack*))
           '(nil)
         (progn
           (setf ,ret-val
             (multiple-value-list
              (progn
                ,@body)))
           (if ,old-within-recording
               (values ,ret-val)
             (process-recordings)))))))

(defmacro test-recording ()
  `(with-recording (n-p)
     (if n-p
         (do-something-and-record 11)
       (do-something-and-record 1))
     (with-recording (n-p)
       (do-something-and-record 2)
       (with-recording (n-p)
       (if n-p
           (do-something-and-record 3)
         (do-something-and-record 33))))))

(test-recording)
;; -> (3 2 1)
```

图 9：重写后的线性 `with-` 宏——感谢 `progv`

对于 *var*。否则，如果 alters-*var* = NIL，那么该形式等价于 (progv nil (list val))，使 *var* 保持不变。

因此，最后一步是将单个 let（或 let*）“拆分”，并按照刚才描述的方式，改用 progv 重新建立那些“有问题”的特殊绑定。由于这一步很难模板化，我们不妨直接看图 9 中最终重写后的示例宏。从测试调用可以看出，它的行为正确，而且展开结果显然是线性的。

尽管建立上下文的宏通常不会用于获取其返回值，但仍然建议兼顾这种情况，因此我们在图 9 中也这样做了。检查 System X 这样的大型系统源代码中该宏的所有用例，以识别此类情况，所花的时间比从一开始就正确处理这些情况更多。因此，图 9 中重写后的宏也会返回与原宏相同的值（使用 multiple-value-list 和 values）。

5 技术的有效性

我们统计了对 System 进行完整重新编译期间发生的宏函数调用（“宏展开调用”）次数。

```lisp
(defmacro with-bad-recording-v3 ((control-p) &body body)
  `(if *within-recording*
       (progn
         ;; when 2 is top of the stack, set the flag
         (when (eq (first *recording-stack*) 2)
             (setf ,control-p t))
         ,@body)
     (let* ((*recording-stack* nil)
            (*within-recording* t)
            (,control-p nil))
       ,@body
       (process-recordings))))

(defmacro test-bad-recording-v3 ()
  `(with-bad-recording-v3 (control-p)
     (if control-p
         (do-something-and-record 11)
       (do-something-and-record 1))
     (with-bad-recording-v3 (control-p)
       (do-something-and-record 2)
       ;; if we push 2, then regardless of the
       ;; lexical nesting context,
       ;; 3 is required next on the stack
       (setf control-p t)
     )
     (with-bad-recording-v3 (control-p)
       (if control-p
           (do-something-and-record 3)
         (do-something-and-record 33)))))

(test-bad-recording-v3)
```

图 10：无法使用上述技术重构的宏

X，并比较了原始版本与经过 PROGV 重构的版本之间的结果。[3]

对于原始版本，我们统计了三个关键指数宏的 $12810 + 2431 + 2432 = 17679$ 次调用。

重构版本的调用次数为 $882+530+531 = 1466$ 次，两者的比率为 $17679/1466 = 12.05$。参照第 2 节中的 m^n 符号，我们有 $m = 2$（两个，@body），因此可以假设平均嵌套深度约为 $log_2 12.05 \approx 3.6 = n$。

观察到的最大 FASL 文件大小缩减，是某个文件从 53 2 26 5 $log_2 26.5 \approx$ MB 缩小到仅 4.7 MB——缩小了 倍！这是因为我们可以假设其中对指数宏的使用嵌套得更深。

这些只是粗略估算，但数据清楚地表明：对于大型 Lisp 系统，即使是嵌套深度适中（即 $3 \le n \le 5$）的指数级宏，也可能严重影响编译时间和 FASL 文件大小。

6 技术的局限性

虽然这些重构模式在实践中应能覆盖大量指数型 with-case，但它们远不能提供完整的解决方案。图 10 展示了一个无法用目前讨论过的技术重构的宏示例。本质上，问题在于，*else* 分支为 control-p 建立了词法上下文，而*then* 分支没有，并且

[3]显然，基于 FLET 的技术会产生相同的结果 s.

不可能在编译时知道哪个分支会被执行。我们需要一种类似 progv 的解决方案，但适用于词法变量。

7 结论

我介绍了将指数级宏重构为线性宏的技术。根据我使用 System X 的经验，我了解到，三个（设计得并不十分严谨的）宏就足以严重地（即呈指数级地）影响编译时间和 FASL 文件大小。

如果因为这些宏在 System X 后期的生命周期中造成了如此多的问题，就责怪最初设计它们的人员，这并不完全公平。每个项目都是从小规模开始的，而这些宏最初运行得很好。直到 System X 生命周期的后期，指数级宏展开的影响才使其编译时间（以及相应的 FASL 文件大小）急剧恶化。增量开发和编译、夜间构建系统、多名开发人员，以及注重功能性而非非功能性系统特性，都是促使代码像恶性癌症一样不断增长的因素。Common Lisp 宏的强大能力可能是一把双刃剑，需要谨慎而细致地使用。幸运的是，Common Lisp 也足够强大，能够为这些问题提供解决方案，正如我们在这篇经验报告中试图说明的那样。我们希望我们的经验能帮助其他开发人员在自己的项目中避免出现类似情况。

这一重写过程能否实现部分自动化？当然可以，编译器可以对可能代价高昂的宏展开发出警告，或者尝试识别指数级展开。作为 Common Lisp 集成开发环境的一部分，macroexpand-all 也肯定会有所帮助。有趣的是，第 3 节和第 4 节中描述的代码重写技术似乎足够简单明了，或许可以实现自动化，至少对于某些宏模式而言（但一般情况下可能无法判定）。这可能是一个值得关注的未来研究方向；我也很乐意收到 Lisp 社区的任何建议和反馈——这个问题想必并非新问题，但我没能找到涉及这一主题的论文。我希望这份报告能填补这一空白，并提高 Common Lisp 开发人员对这类问题的认识。

致谢

首先，我要感谢 Project X 的首席研究员（PI）允许我为 System X 提供咨询，并促使我对 System X 中编译时间过长的问题展开调查。这位 PI 还阅读了一份草稿，并建议通过确切的数值证据来佐证研究结果。因此，我们新增了第 5 节。

接下来，我要感谢匿名评审指出原始草稿中的遗漏。这些遗漏使得从基于局部函数调用的方案（如 with-good 宏所示）过渡到基于 progv 的方案显得缺乏动机。正如评审所指出的，原始草稿确实提到了这项技术，但错误地让人以为它无法用于适配词法变量。现在第 3 节展示了这一点，而该节在原始草稿中并不存在。感谢所有评审提出意见并付出的时间。

依我拙见，这两种重构方法各有优缺点，我在第 3 节末尾提到过其中一些。

本研究由美国国立卫生研究院下属的国家过敏症和传染病研究所通过资助编号 R01AI160719 提供经费支持。本文内容完全由作者负责，不一定代表美国国立卫生研究院的官方观点。美国国立卫生研究院未参与本研究的设计、数据收集、分析或解释，也未参与论文的撰写。

参考文献

[1] Marco Baringer。访问日期：2023 年 2 月 26 日。ARNESI 库。https://bese.common- lisp.dev/arnesi.html [2] Marco Baringer。访问日期：2023 年 2 月 26 日。ARNESI 库。https://quickref.common-lisp.net/arnesi.html [3] Daniel G. Bobrow、Kenneth Kahn、Gregor Kiczales、Larry Masinter、Mark Stefik 和 Frank Zdybel。1986。《CommonLoops：融合 Lisp 与面向对象编程》。载于面向对象编程系统、语言与应用会议论文集（美国俄勒冈州波特兰）（*OOPSLA '86*）。计算机协会，纽约州纽约，17–29。https://doi.org/10.1145/ 28697.28700[4] P. Graham。1994 年。《论 *Lisp*》。Prentice-Hall。[5] P. Graham。1996 年。《*ANSI Common Lisp*》。Prentice-Hall。[6] Ralf Möller、Michael Wessel。2008 年。描述逻辑系统的软件抽象。载于第五届欧洲 *Lisp* 工作坊论文集（*ELW'08*）。http://www. european-lisp-workshop.org/archives/08.wessel.pdf [7] P. Norvig。1992 年。《人工智能编程：*Common Lisp* 案例研究》。Morgan Kaufmann。[8] Guy L. Steele, Jr. 1990 年。《*Common Lisp*——语言，第二版》。Digital Press。[9] Michael Wessel。2022 年。基于本体的虚拟个人助理对话管理框架：使用 Common Lisp 实现。载于*2023* 年欧洲 *Lisp* 研讨会（*ELS'23*）。https://doi.org/10.5281/zenodo.6335631