Notes on Refactoring Exponential Macros in Common Lisp
Or: Multiple `@Body` Considered Harmful

Michael Wessel
michael.wessel@sri.com
SRI International
Menlo Park, California, USA

ABSTRACT

I recently consulted for a very big Common Lisp project having more than one million lines of code (including comments). Let's call it "System X" in the following. System X suffered from extremely long compilation times; i.e., a full recompile took about 33:17 minutes on a 3.1 GHz MacBook Pro Intel Core i7 with SSD and 16 GBs of RAM, using ACL 10.1. It turns out that a number of macros were causing an exponential code blowup. With these macros refactored, the system then recompiled in 5:30 minutes — a speedup by a factor of ≈ 6. In this experience report, I will first illuminate the problem, and then demonstrate two potential solutions in terms of macro refactoring techniques. These techniques can be applied in related scenarios.

KEYWORDS

Common Lisp, Macros, Exponential Code Blowup, Macro Refactoring, Very Large Lisp Systems

1 INTRODUCTION

Macros and the ability to program language extensions in the language *itself* is one of the most beloved and powerful features of many members of the Lisp family, and especially in Common Lisp [8], which has been coined a *programmable programming language* by John Foderaro. The availability of the full programming language at macro expansion / compile time makes Common Lisp an ideal implementation platform for Domain Specific Languages [6, 9], and always has been (i.e., Lisp was an early target platform for object-oriented programming concepts [3]). Unlike macros in most other programming languages, Common Lisp allows macros to be defined in the same language. Thanks to its *homoiconicity*, it offers a unified "programs as data" representation and allows the construction, manipulation, and most importantly, *computation* of macro expansions in the language *itself.* The full power of the language is always available – not only at runtime, but also at *macro-expansion ("compile") time* [4, 5, 7].

As always, with great power comes great responsibility: macros can be a double-edged sword. This is especially true in languages like Common Lisp, where the main development mode is not the traditional "edit — full recompile – debug" cycle, but an interactive, dynamic one, based on incremental redefinition, evaluation, and compilation. Unintended consequences of changes to the code base, especially macros, can sometimes be left unnoticed for a longer time period if full recompiles of the system are delayed. This holds true especially in larger projects with bigger teams. Once compilation times exceed half an hour, full recompilation is avoided by the developers during daily development, and a build system will usually be entrusted to deliver new base images overnight, containing the changes of multiple developers. Of course, regressions will be recorded and monitored on a daily basis. But even if *build times* and *the size of the fast load (FASL) files* are reported by the build system, it might not be entirely clear which changes increased the build time — after all, the build system might just have had a bad night and was busy performing backups as well, and so on and so forth.

Consequently, tracing back unintended system behavior to (no longer so recent) changes to the code base can become more difficult. For this reason, incremental compilation of Common Lisp code can become a drawback. I advise that developers should not only check for unintended changes in semantics and functional characteristics of the system caused by their code changes, but also to the non-functional characteristics (e.g., FASL sizes and build time). And especially for macros.

I recently had the opportunity to work on "System X", which is a very large, multiple decades in-the-making Common Lisp system with over one million lines of code. System X suffered from extremely long compilation times: a full compile required 33:17 minutes on a 3.1 GHz MacBook Pro Intel Core i7 using an SSD and 16 GBs of RAM, using ACL 10.1. It turned out that *three badly engineered macros* were causing an exponential code blowup. With these macros refactored, a full recompile is possible in 5:30 minutes — a speedup by a factor of ≈ 6.

In this experience report, I will first illuminate the problem, and then demonstrate two possible solutions in terms of macro refactorings. The effectiveness of the refactoring methods is not only demonstrated by the 6 fold reduction in compilation time, but also in terms of FASL size reductions. The used methodology can be applied in related scenarios. I conclude with some advice.

2 THE PROBLEM

The problems of System X are easily illustrated with a few synthetic examples. Consider the context-establishing `with-bad` macro in Figure 1. Like many `with-` macros, it utilizes a *special variable* (here, `*a*`) with dynamic scope to control the runtime behavior beyond its lexical scope. In the example, the binding of the dynamic variable `*a*` determines if `with-bad` macro's expansion prints `x` or `(1+ x)`.

This macro serves to illustrate the problem of *exponential macro expansion.* Frequently, `with-` macros are nested, which can obscure the magnitude of such problems from developers. For example, the macro might be part of a framework for website HTML generation and, as such, contain macros such as `with-head`, `with-body`, `with-table`, and so on. Not only will complex web pages contain many deeply nested occurrences of these macros, but it might also be the case that certain common design elements of such pages

```
(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)))))
```

```
(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))))))))
```

Figure 1: Macro with exponential macro expansion

```
(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)))))
```

```
(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))))
```

Figure 2: Macro with linear macro expansion

(common headers, footers, and navigation menus) have been aggregated into even higher-level macros, which are then being used in other macros, and so forth.

It should be noted that Common Lisp does not contain a `macroexpand-all` recursive macroexpansion facility; only `macroexpand-1` is offered. This provides a single level of macro expansion. However, third-party solutions are available. I used one of these packages to diagnose the problems in System X [1, 2].

Considering the macroexpansion of `blowup` containing four nested `with-bad` occurrences in Figure 1, we can clearly see that it is exponential in the size of the original definition, due to the duplicated `,@body` forms. In general, given a nesting depth of n, the size of the expanded macro code is 2^n. I even spotted `with-` macros with more than two `,@body` forms "in the wild"; in general, a `with-` macro with m `,@body` forms will expand to size m^n if nested n times. This should clearly be avoided.

Sometimes, such an *exponential macro* is easy to fix. In the case of Figure 1, it suffices to move the `,@body` form into a local function definition (`flet`) and call the local function in the two places at runtime rather than duplicating the code. The expansion size of the resulting `with-good` macro shown in Figure 2 is now linear in the nesting depth rather than exponential.

Unfortunately, exponential macros are not always easily fixed. For example, say the macro argument `x` in `with-good` was used to establish lexical bindings for use *within* the `,@body` instead of just being an "input parameter" to the macro. In this case, a `(let ((,x ...)) ... ,@body)` would be used within the macro to establish a corresponding lexical scope for `,x`. Moreover, the concrete binding to `,x` might depend on complex runtime and compile time conditions. In particular, the value of `,x` might depend on the runtime value of `*a*`, which is unknown at compile time / macroexpansion time, and hence, cannot be anticipated by means of code rewritings / transformations. It is thus important that the correct lexical contexts are established, for example, via the local function's lambdalist.

Of such "more difficult" nature was the exponential macro that I had to refactor in System X. Instead of revealing the details of this macro I will use the synthetic example from Figure 3 in the following. This macro has a similar complexity and serves to illustrate the problems and possible solutions.

```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)
```

Figure 3: The anti-pattern of an exponential `with-` **macro as found in System X**

The idea behind `with-bad-recording` is to establish a context of dynamic scope for keeping track of "instructions" that are being recorded onto a stack; these instructions can be entries to a log file, an output recording presentation history, etc. The "hidden" special variable `*recording-stack*` (with dynamic scope) is used to keep track of the values on the stack. This special variable is not supposed to be visible to the user's code (it is "internal"); instead, accessor functions (or macros) such as `(do-something-and-record x)` are used to work with it.

Moreover, for whatever reason, clients of `with-bad-recording` also like to know whether the current invocation is toplevel, or already part of a "nested" invocation at runtime; hence, a variable `nested-p` can be passed in which is then bound to `nil` or `t`, respectively. To decide this, another special variable `*within-recording*` had been (maybe redundantly) introduced by the original author of the macro. Again, this is an "internal" special variable which should not be visible to the user code, hence, `nested-p` is supplied. A use case is shown in `test-bad-recording`. Note that the runtime value of `n-p` (i.e., `,nested-p` in the macro) is not knowable from the lexical definition, as `test-bad-recording` might occur nested within another `with-bad-recording` context at runtime. The use case shows that the stack holds `(3 2 1)` in the end.

```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.
```

Figure 4: Replacing multiple `,@body` **with local function calls broke the macro — see the compiler warning**

Clearly, this macro now has the potential for an exponential macro expansion, and `test-bad-recording` already suffers from this blowup. Can we fix this macro in the same way as in Figure 2?

3 FIRST SOLUTION — REFACTORING WITH FLET

The first solution is a generalization of the idea we already discussed in Section 2 — we replaced each `,@body` by a call to a single local function containing a single `,@body`. We can use `FLET` or `LABELS` for that.

A naive attempt of fixing `with-bad-recording` is shown in Figure 4. This macro is now clearly broken, as `,@body` refers to `,nested-p = n-p`, which is not visible in the outer `flet` - hence the compiler warning that this variable is now unbound. The obvious solution is hence to make `,nested-p = n-p` an argument of the local function so that the required lexical variables for `,@body` are established by the local function. This is shown in Figure 5.

A further complication is introduced if the lexical variable is modified in one of the branches — consider the variation `with-bad-recording-v2` shown in Figure 6, where `nested-p` is replaced by `control-p`. The value of `control-p` influences the output, and it might be set from either within the user-supplied `,@body` code, or from within the macro itself. Refactoring such a macro then becomes less mechanical, and more care is needed to ensure that the right lexical environments are established.

```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)
```

Figure 5: Using local function variables to establish lexical context — note that the dynamic context is still established by the original branching structure

```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)
```

Figure 6: If lexical variables that are arguments to the macro are modified, such as `control-p`, then refactoring becomes more involved

In particular, we realize that the value of `control-p` must be changed from *within* the local function so that the call

```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)
```

Figure 7: Accommodating different macro branches in one function with the `branch-p` argument

`(process-recordings-v2 ,control-p)` will get the value of `,control-p` from the correct lexical scope. A possible solution is shown in Figure 6; the `branch-p` argument is used to inform the local function about the invocation context, and, based on its value, the local function has to accommodate, or "emulate", the different runtime behaviors from the macro's original branches. This might not always be possible, but is rather trivial in this case.[1]

This wraps up the discussion of the first refactoring strategy. In a nutshell, the original branching structure establishing different lexical and dynamic scopes is maintained. A common `,@body` form must be found which is able to reproduce the original runtime behaviors, and it is placed within a local function. The required lexical contexts are established by the local function and removed from the original branching structure. Additional control parameters such as `branch-p` are used to select branch-specific runtime behavior. In particular, we maintain the branching structure of the original macro in order to establish the right bindings for the special variables, and to set up the correct lexical contexts by calling the local function accordingly.

A potential drawback of the refactoring pattern just discussed is the introduction of additional local functions and the additional runtime overhead of additional function calls.[2]

More severely, the (full) macro expansion of the refactored macro now obfuscates the original structure of the macro — it is "inside out" because we employed *functional composition* to implement *program sequencing;* as can be seen in Figure 2, `(princ 6)` now textually precedes `(princ 3)`, contrary to the original definition. In particular, the `,@body`'s of the local functions are now "detached" from the original branching structure, making the macro expansion more difficult to understand.

If these are serious concerns, the following alternative refactoring strategy can be applied.

[1]Please note that we are ignoring potential differences in the returned value of these macros for now; usually, `with-` macros do not return values, but this is a convention and not a strict requirement.

[2]It might be possible to declare these local functions as `inline` though.

4 SECOND SOLUTION — REFACTORING WITH PROGV

In the following, we are *not* using a local function that can be called from different branches of the rewritten macro. Instead we are trying to unify the original branching structure establishing different lexical and dynamic contexts into *one* common structure. It turns out that establishing the right (conditional) bindings for the special variables is the biggest obstacle, and we will be using progv for this purpose.

The following set of steps can be understood as a semantics-preserving code rewriting procedure / transformer. We will apply the following to the with-bad-recording macro to tame the exponential beast and rewrite it into a linear macro:

Step 1 Macroexpand / rewrite all branching special forms (unless, when, cond, ...) into ifs (in our examples, this is already the case, so the step doesn't apply):

```
(if <condition>
     (let ( <binding 11> ... <binding 1n> )
         ,@body)
   (let ( <binding 21>  ... <binding 2m> )
       ,@body))
```

Step 2 Ensure that all lets in all branches refer to the same variables, and in the same order. If <binding ij> = (,var val) and var is a macro argument, then all branches already must contain a valid (,var val) binding. Otherwise, var would be unbound in (some branches of) ,@body (e.g., the macro was already defective in the first place).

If var is a special variable instead, i.e., *var*, then, in case the branch *did not* contain a <binding> = (*var* val), we introduce a "dummy" binding <binding> = (*var* *var*) for now. The idea is to express that we *intend* to *not* alter the binding of *var* dynamically. Note that this is *unproblematic* where *var* is used as a "read only" variable, but problematic in cases such as with-bad-recording, where *recording- stack* is modified; see below for the solution.

Hence, we now have the same number k of (var val) bindings in each let, with potentially (not necessarily) different val's; note that $max(n, m) \le k \le n + m$:

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

Step 3 Next, we remove the different branches, establish all the bindings in a single let, and recover the effects of the <condition> by establishing different bindings within the let binding forms itself. Since we removed the different branches from the surrounding code by factoring in / moving the condition *into* the let lambda lists, we have also eliminated the multiple ,@body occurrences:

```
(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)
```

Figure 8: Shadowing *recording-stack* broke the macro

Step 4 So far so good — there is one problem though: this only works for dynamically scoped variables that are used in a "read only" fashion. As already mentioned, we have introduced a "dummy" binding <binding> = (*recording-stack* *recording-stack*) to express that we wish to leave the binding of *recording-stack* *untouched.* But we changed it by establishing a new binding frame — we "shadowed" the previously established binding. With let/let*, there is no solution to this.

The effect is illustrated in Figure 8 — the refactored macro is clearly broken now, as illustrated with the example call (test-recording-v4). Instead of returning (3 2 1) like in the original, we are now only getting the first value that was pushed onto the stack: (1).

However, the code rewritings have brought us onto the right track. We only need to avoid shadowing in cases were we do not wish to alter a dynamic variable. Fortunately, there is a solution to this in Common Lisp, and one has to congratulate the designers of Common Lisp for anticipating such a scenario: progv can do the job as follows:

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

If alters-*var* = T, the form is equivalent to (progv '(*var*) (list val)) hence establishing a new binding

```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)
```

Figure 9: The rewritten linear `with-` macro — thanks to `progv`

for `*var*`. Otherwise, if `alters-*var* = NIL`, then the form is equivalent to `(progv nil (list val))`, leaving `*var*` unchanged.

Hence, the final step involves "splitting up" the single `let` (or `let*`), and reestablishing the "problematic" special bindings via `progv` instead, in the manner just described. Since this step is hard to templatize, let's look at the final rewritten example macro in Figure 9 instead. As can be seen from the test invocation, it behaves correctly, and its expansion is clearly linear.

Even though context establishing macros are usually not used for their return values, it is nevertheless advisable to accommodate for such, and so we did in Figure 9. Inspecting all use cases of the macro in the source code of a very large system such as System X to identify such use cases is more time consuming than to cater for such cases correctly from the beginning. Hence, the rewritten macro in Figure 9 also returns the same values as the original (utilizing `multiple-value-list` and `values`).

5 EFFECTIVENESS OF THE TECHNIQUES

We counted the number of macro function invocations ("macro expansion calls") that occurred during a full recompilation of System X and compared the results between the original and the PROGV refactored versions.[3]

For the original version, we counted $12816 + 2431 + 2432 = 17679$ calls for our three critical exponential macros. Compared to $882 + 530 + 531 = 1466$ invocations for the refactored version, with a ratio of $17679/1466 = 12.05$. Referring to the m^n notation from Section 2, we have $m = 2$ (two `,@body`'s), and can hence assume an average nesting depth of about $log_2 12.05 \approx 3.6 = n$.

The biggest FASL size reduction was observed for a file that shrank from 53 MBs to only 2 MBs — a factor of 26.5! Since $log_2 26.5 \approx 4.7$ we can assume a more deeply nested use of the exponential macros there.

These are rough estimates, but the numbers speak a clear language: for large Lisp systems, the impact of even moderately deeply nested (i.e., $3 \leq n \leq 5$) exponential macros can be catastrophic in terms of compilation time and FASL sizes.

```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)
```

Figure 10: A macro that cannot be refactored using the discussed techniques

6 LIMITATIONS OF THE TECHNIQUES

Whereas these refactoring patterns should cover a large region of exponential `with-` cases in practice, they are far from offering a complete solution. An example of a macro that *cannot* be refactored with the so-far discussed techniques is shown in Figure 10. Essentially, the problem is that the *else* branch establishes a lexical context for `,control-p`, but the *then* branch doesn't, and that it is

[3] Obviously, the FLET-based technique will yield the same results.

impossible to know at compile time which branch will be active. A solution akin to `progv` would be needed, but for *lexical* variables.

7 CONCLUSION

I presented techniques for refactoring exponential macros into linear ones. From my experience with System X, I learned that three (not overly carefully designed) macros can suffice to severely (i.e., exponentially) affect compilation time and FASL size.

It is not entirely fair to blame the original designers of these macros for causing so much trouble in the later life of System X. Each project starts small, and the macros were originally doing fine. Only later in System X's life cycle did the effects of exponential macro expansion degrade its compilation time (and corresponding FASL file sizes) drastically. Incremental development and compilation, over-night build systems, multiple developers, and a focus on *functional* rather than *non-functional* system characteristics were all factors that contributed to code that grew like a malignant cancer. The power of Common Lisp macros can be a double-edged sword and needs to be handled with diligence and delicacy. Fortunately, Common Lisp is also powerful enough to offer a cure to these problems, as we tried to illustrate in this Experience Report. We hope that our experience will help other developers to avoid such situations in their own projects.

Could some of this rewriting process be automated? For sure, compilers could warn about potentially expensive macro expansions, or try to identify exponential expansion. A `macroexpand-all` as part of a Common Lisp IDE would certainly help as well. Interestingly, the code rewriting techniques described in Sections 3 and 4 seems straight-forward enough that it *might be possible* to automate, at least for certain macro patterns (but might be undecidable in general). This could be interesting future research, and I would appreciate any pointers and feedback from the Lisp community — surely, this problem is not new, yet I wasn't able to find papers that would cover this topic. I hope that this report will fill this gap, and also raise awareness in Common Lisp developers for such issues.

ACKNOWLEDGMENTS

First and foremost, I would like to thank the Principial Investigator (PI) of Project X for allowing me to consult for System X, and for triggering the investigation of long compilation times in System X. This PI also read a Draft and suggested to substantiate the findings with hard numerical evidence. This led to the addition of Section 5.

Next, I wish to thank the anonymous reviewers for pointing out omissions in the original Draft that made the transition from the local function call-based solution (as illustrated with the `with-good` macro) to the `progv`-based solution seem unmotivated. As pointed out by the reviewers, the original Draft did mention the technique, but erroneously gave the impression that it could not be applied to accommodate for lexical variables. This is now demonstrated in Section 3, which was missing from the original Draft. I like to thank all reviewers for their comments and time.

In my humble opinion, both refactoring methods have their own strengths and weaknesses, and I mentioned some of them at the end of Section 3.

This work was funded by the National Institute of Allergy and Infectious Diseases of the National Institutes of Health under award number R01AI160719. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health. The NIH did not play any role in the design of the study; nor in collection, analysis, or interpretation of data; nor in writing the manuscript.

REFERENCES

[1] Marco Baringer. Accessed: 2-26-2023. ARNESI Library. https://bese.common-lisp.dev/arnesi.html
[2] Marco Baringer. Accessed: 2-26-2023. ARNESI Library. https://quickref.common-lisp.net/arnesi.html
[3] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. 1986. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '86)*. Association for Computing Machinery, New York, NY, USA, 17–29. https://doi.org/10.1145/28697.28700
[4] P. Graham. 1994. *On Lisp*. Prentice-Hall.
[5] P. Graham. 1996. *ANSI Common Lisp*. Prentice-Hall.
[6] Ralf Möller Michael Wessel. 2008. Software Abstractions for Description Logic Systems. In *Proceedings of the 5th European Lisp Workshop (ELW'08)*. http://www.european-lisp-workshop.org/archives/08.wessel.pdf
[7] P. Norvig. 1992. *Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann.
[8] Guy L. Steele, Jr. 1990. *Common Lisp – The Language, Second Edition*. Digital Press.
[9] Michael Wessel. 2022. An Ontology-Based Dialogue Management Framework for Virtual Personal Assistants in Common Lisp. In *European Lisp Symposium '23 (ELS'23)*. https://doi.org/10.5281/zenodo.6335631