Dynamically Scoped Functions as the Essence of AOP

Pascal Costanza
University of Bonn, Institute of Computer Science III
Römerstr. 164, D-53117 Bonn, Germany
costanza@web.de, http://www.pascalcostanza.de

June 17, 2003

Abstract

The aspect-oriented programming community devotes lots of energy into the provision of complex static language constructs to reason about eventual dynamic properties of a program. Only the focus on a strongly dynamically-natured language construct reveals a very simple mechanism behind the notions of AOP. This paper shows that dynamically scoped functions, together with a simple additive to access previous function definitions, form the essence of aspect-oriented programming. We present a complete executable one page implementation of this idea.

1 Lexical vs. Dynamic Scope

A definition is said to be dynamically scoped if at any point in time during the execution of a program, its binding is looked up in the current call stack as opposed to the lexically apparent binding as seen in the source code of that program. The latter case is referred to as lexical scoping. Consider the following program in Scheme [12]:

```
(define factor 3.14)

(define (f x) (* factor x))

(define (g x)
  (let ((factor 1.96))
    (f x)))
```

Since in Scheme definitions are always lexically scoped, functions `f` and `g` always return the same results for the same arguments. The variable `factor` is rebound in `g`, but this does not affect the behavior of `f` since `f` always sees the lexically apparent binding of `factor`. Compare this to the following similar program in Common Lisp [1]:

```
(defvar *factor* 3.14)

(defun f (x) (* *factor* x))

(defun g (x)
  (let ((*factor* 1.96))
    (f x)))
```

In Common Lisp, a global variable introduced by `defvar` is always dynamically bound. In order to be able to distinguish them from lexically bound variables, they are given names that have leading and trailing asterisks. This is an idiom generally accepted by Common Lisp programmers. In Common Lisp, dynamically scoped variables are referred to as *special* variables [1, 18, 19].

In the program above, `f` yields different results from `g` for the same arguments, since the new binding of `*factor*` in `g` affects the behavior of `f`. Dynamically scoped variables are always looked up in the call stack starting from the current method frame. When `g` is called, the first binding that is found for `*factor*` within `f` is the one introduced by `g`; otherwise, when `f` is directly called, it is the global one.

Almost all programming languages in wide use employ lexical scoping but do not offer dynamic scoping. For example, the following program in Java [8] is equivalent in this respect to the Scheme version above:

```
public class Test {

  public static double factor = 3.14;

  public static double f (double x) {
    return factor * x;
  }

  public static double g (double x) {
    double factor = 1.96;
    return f(x);
  }
}
```

1.1 Origin and Uses of Dynamic Scope

Dynamic scoping was accidentally introduced in the first implementations of Lisp back in the 1950's. When Lisp was implemented as an interpreter, dynamic scoping just happened to be the most natural implementation for variable lookup. Compilation of Lisp naturally led to lexical scoping. The lack of understanding of the issues involved was ultimately remedied by the introduction of proper lexical scoping into the Scheme dialect of Lisp in the 1970's [20]. A detailed discussion of the issues involved around scoping can be found in the seminal technical reports [21] and [23].

As already pointed out in those reports, dynamically scoped variables turn out to be very useful when there is a need to influence the behavior of parts of a program without having to clutter the parameter lists of the functions called directly and/or indirectly. For example in Common Lisp, there are several standardized special variables that control the behavior of the standard print functions. One example is `*print-base*` that specifies the radix to use when printing numbers. So by binding `*print-base*` to, say, 16 one can ensure that all numbers are printed in hexadecimal representation, no matter where the printing takes place below a certain point in the call stack. Consider the following program fragment:

```
(let ((*print-base* 16))
  (dotimes (i 20)
    (dotimes (j 20)
      (print i)
      (print j))))
```

The `i`-s and `j`-s are all printed in hexadecimal representation although this is not made explicitly clear at each invocation of `print`. It should be clear by now that this would also be the case for arbitrarily nested function calls inside the `dotimes` loops that directly or indirectly call `print`.

The idiomatic use of dynamically scoped variables in Common Lisp balances several forces:

- The naming convention for special variables ensures that local lexically scoped variable declarations never accidentally capture global dynamically scoped variables. In the following code fragment it is very clear which variable is dynamically scoped and which one is lexically scoped:

  ```
  (defvar *factor* 3.14) ; dynamically scoped

  (defun f (x)
    (let ((factor 1.96)) ;   lexically scoped
      ...))
  ```

- Common Lisp does not provide for lexically scoped global variables – one can just refrain from rebinding special variables. Again, local variables cannot accidentally conflict with global variables because of the naming convention for those global variables. In those rare cases in which a lexically scoped global variable is actually needed, because it is important to be able to rebind it lexically, `define-symbol-macro` can be used to emulate it. See [1, 18] for more information on `define-symbol-macro`; see the chapter on continuation-passing macros in [9] for an example that makes use of a lexically scoped global variable.

- Common Lisp provides means to declare dynamically scoped local variables. This is only useful in rare circumstances, and fortunately the language specification ensures that it is not possible at all to accidentally introduce such variables. See [3] for an example that makes use of dynamically scoped local variables.

2 Dynamically Scoped Functions

Dynamically scoped functions had been discussed as a possible addition to Common Lisp during the ANSI standardization process, but an agreement was not reached.[1] The following forces need to be considered in the design of a language construct for dynamically scoped functions:

- If by default all functions were dynamically scoped, unintentional conflicts between function names would be unavoidable. So there needs to be a useful way to discriminate between lexical and dynamic scoping for functions.

- A naming scheme for discriminating between lexically and dynamically scoped functions like that for special variables in Common Lisp - leading and trailing asterisks - would be very unhandy, because there are usually considerably more global function definitions than global variables.

- Furthermore, it is not clear whether the decision for lexical or for dynamic scope should really be made alongside the (global) function definitions or somewhere else. After all, it is not yet clear what dynamically scoped functions would be good for.

The goal of this paper is to provide a full-fledged aspect-oriented language construct. In order to achieve this, we propose the following approach for dynamically scoped functions in Common Lisp:

- The decision for lexical or dynamic scope is to be made alongside the local definition of a function.

[1] Thanks to Kent M. Pitman for this piece of information.

Common Lisp already has the `flet` form for defining lexically scoped local functions. We add `dflet` for rebinding a global function with dynamic extent. Consider the following program fragment:

```
(defun f (x) (print x))

(defun g (x) (f x))

(flet ((h (x) (g x)))
  (dflet ((f (x) (print (+ 1 x))))
    (h 5)))
```

Here, `h` is called within a dynamically scoped definition of `f`. Therefore, the subsequent indirect call of `f` within `g` executes that definition. Hence, this program fragment prints `6`, not `5`.

- A construct for dynamically scoped function definitions is not enough. We also provide a way to refer to the previous definition of a function by way of an implicit local `call-next-function` definition. The program fragment above can thus be rewritten as follows:

```
(flet ((h (x) (g x)))
  (dflet
    ((f (x) (call-next-function (+ 1 x))))
      (h 5)))
```

The `call-next-function` is reminiscent of CLOS's `call-next-method` [1, 18, 19] and `proceed` in AspectJ [13]. Now, dynamically scoped functions can be used to implement the canonical example of aspect-oriented programming:

```
(defun f (x) (print x))

(dflet ((f (x) (print "entering f")
               (call-next-function)
               (print "leaving f")))
  (f 5))
```

The output of this program fragment is as follows:

```
"entering f"
5
"leaving f"
```

So indeed, dynamically scoped functions model the essence of aspect-oriented programming:

- A `dflet` captures specific join points in the control flow of a program, and its definitions cross-cut the program at each invocation of the original functions.
- A `dflet` has dynamic extent: As soon as the control flow enters a `dflet`, the new function definitions are activated and on return, they are deactivated again. This is opposed to the static "weaving" approach typically employed in AOP.
- So with `dflet`, there is no need to add constructs for statically reasoning about the control flow of a program (for example, `cflow` and friends in AspectJ [13]).

2.1 Notes on Implementation

Appendix A lists a complete implementation of the ideas outlined so far in Common Lisp. Dynamic scoping for function definitions is accomplished by reusing Common Lisp's special variable machinery. The body of a function is stored in a special variable and at the same time, the function to be called just forwards any call to the one stored in that special variable. Consider the following function definition:

```
(defun f (x) (* x x))
```

In principle, the implementation in Appendix A translates this into the following definitions:

```
(defvar *f* (lambda (x) (* x x)))

(defun f (&rest args) (apply *f* args))
```

In order to avoid unintentional name clashes, the symbol to be used for naming the special variable (`*f*` in the example above) is generated programmatically and stored in the hash table `*dynsyms*` that maps function names to such symbols (lines 1-6 in Appendix A).

Since we do not want to replace the standard `defun` macro in Common Lisp, we provide our own `defdynfun` (lines 12-28). Changing `defun` would not be advisable because of Common Lisp's facility to declare functions to be inlined in compiled code. There is no standard way to modify such inlined functions after the fact, so a dynamically scoped definition would not be able to affect inlined functions. Therefore it should be a conscientious decision by a programmer whether a function may have dynamically scoped definitions or not. Since this requirement should not result in a too strong restriction, we also provide a way to turn a `defun`-ed function into a `defdynfun`-ed one after the fact via `redefdynfun` (lines 27-28). So the following two definitions are effectively equivalent:

```
a) (defdynfun f (x) (* x x))

b) (defun f (x) (* x x))
   (redefdynfun f)
```

```
(let (memo)
  (dflet ((fib (x) (let ((result (cdr (assoc x memo))))
                     (if result
                         (prog1 result
                           (format t "~& found (fib ~A) => ~A" x result))
                         (prog1 (setf result (call-next-function))
                           (format t "~& called (fib ~A) => ~A" x result)
                           (setf memo (acons x result memo)))))))
    (loop (print '>) (print (eval (read))))))
```

Table 1: A dynamically scoped redefinition of the fibonacci function that caches computations.

If a `dflet` attempts to define a function that is not properly prepared, an error handler allows the programmer to correct this situation (lines 32-37). If `redefdynfun` is applied to a function name that does not exist yet, a function with that name is prepared for dynamically scoped definitions nonetheless, which in turn signals an error on invocation as its default behavior (and thus behaves like any other undefined function; see lines 20-24).

The `dflet1` helper macro allows for the dynamically scoped definition of exactly one function, whereas the `dflet` macro allows for zero or more definitions and uses `dflet1` repeatedly.

(Common Lisp's `(reduce ... :from-end t)` is better known to the world as foldr.)

The `ensure-dynsym` and `dflet1` macros make use of `with-gensyms` as described in the chapter on classic macros in [9]. Apart from that, the source code uses only forms as defined by ANSI Common Lisp [1, 18]. The code has been developed and tested with Macintosh Common Lisp 5.0 (http://www.digitool.com) and the beta version of LispWorks for Macintosh 4.3 (http://www.lispworks.com).

2.2 An Example

Following the tradition to illustrate aspect-oriented programming with caching of fibonacci numbers, we show how to implement this in our approach. First we give a function definiton for fibonacci numbers, CLOS-style, and declare it to be overridable by `dflet`:

```
; methods for the "objects" 0 and 1
(defmethod fib ((x (eql 0))) 1)
(defmethod fib ((x (eql 1))) 1)

; a method for all other cases
(defmethod fib (x)
  (+ (fib (- x 1))
     (fib (- x 2))))

(redefdynfun fib)
```

A dynamically scoped definition of `fib` allows us to cache the computations – see Table 1 above.

The technique used here is essentially the same as that of [17]: Results are looked up in an association list. When they are found, they are immediately returned; otherwise the original function is called, the result is stored in the association list and finally returned. All this is interspersed with verbose output. The body of `dflet` is a loop that allows interactive invocations of `fib` and observation of the caching behavior.

Here is a transcript of an example session:

```
> (fib 4)
 called (fib 1) => 1
 called (fib 0) => 1
 called (fib 2) => 2
 found (fib 1) => 1
 called (fib 3) => 3
 found (fib 2) => 2
 called (fib 4) => 5

5

> (fib 4)
 found (fib 4) => 5

5
```

3 Related Work

3.1 Dynamic Scope

Scheme does not provide any standard constructs for dynamic scoping - in [21], an implementation of dynamically scoped variables on top of lexical scoping is presented, and the reports that define the Scheme standard head for minimalism and conceptual simplicity instead of completeness.

However, many Scheme dialects, for example Mz-Scheme [6], provide dynamic scope in two ways: as the `fluid-let` form, and as parameter objects. For example,

the idea presented in this paper can be implemented with `fluid-let` roughly as follows:

```
(define (f x) (print x))

(let ((call-next-function f))
  (fluid-let ((f (lambda (x)
                   (print "entering f")
                   (call-next-function x)
                   (print "leaving f"))))
    (f 5)))
```

A major drawback of `fluid-let` is that it is explicitly defined to save a copy of the previous value of a global variable on the stack and establish the new binding by side-effecting that variable. This implementation breaks in the case of multi-threading - each thread should have their own independent bindings for dynamically scoped variables instead of randomly modifying shared storage. In fact, Common Lisp implementations that incorporate multi-threading typically treat special variables as intuitively expected.

This is also true for parameter objects in MzScheme: a parameter object can be accessed as a function that either gets or sets its current value, and it can be given a new binding with dynamic scope in a thread-safe manner by way of the `parameterize` form. However, the fact that parameter objects can only be accessed via functions together with Scheme's preference of macro hygiene requires a little bit more gymnastics for an implementation of our approach.

Recent attempts at introducing dynamically scoped variables into C++ [10] and Haskell [16] would in principle also be suitable for our approach because they are also implemented by passing dynamic environments to functions behind the scenes, instead of modifying global state. However, we have not yet worked out the details in either case.

Emacs Lisp comes with a Common Lisp compatibility package that differs from ANSI Common Lisp for example with regard to local functions: `flet` is lexically scoped in ANSI Common Lisp, but dynamically scoped in that package [7]. This means that an implementation of our approach would be straightforward in Emacs Lisp.

It is interesting to note that many programming languages provide dynamically scoped functions in the form of exception handlers: a throw of an exception can be understood as a call to the dynamically scoped exception handler associated with that exception. However, exceptions are not especially useful for aspect-oriented programming because throws of exceptions are too explicit.

3.2 Aspect-Oriented Programming

There have been previous aspect-oriented attempts at providing language constructs for rebinding / amending existing functions at runtime, similar to our approach based on dynamically scoped functions. In [14], the semantics of a superimposition language construct are defined that allows for method call interception (MCI). Like in our approach, method call interceptors can be activated and deactivated at arbitrary points in the control flow of a program and have dynamic extent. However, that paper defines a superimposed method to globally replace the method's original definition during that extent. This would have the same drawbacks as MzScheme's `fluid-let` in the presence of multi-threading, but can probably be corrected by switching to one of the thread-safe techniques. Approaches like Handi-Wrap [2] and AspectS [11] can be regarded as implementations of the semantics described in [14], and they have the same drawbacks and differences to our approach. In [15], an approach is described how to add superimposition of this kind to any language, provided the semantics of a language are given in a suitable style.

In [24], an aspect-oriented extension of Scheme is described that provides both constructs for static and dynamic weaving (`around` and `fluid-around`). In that approach, `fluid-around` stores and looks up aspect activations in the call stack, and therefore is thread-safe. The static `around` construct records aspects in a global environment and ensures that closures keep their respective static aspects when they escape the scope of an `around` activation. The same degree of "stickiness" can be achieved in our approach by passing around the closure stored in the symbol returned by `get-dynsym` (line 3 in Appendix A), instead of the function itself.

All the dynamic aspect-oriented approaches described above can be regarded as derivations of the idea to run programs inside a specialized "aspect-oriented" monad [17]. However, the former approaches are embedded in the respective base languages while the latter requires all interceptors to be defined in the monad. The fundamental difference to our approach is that all those approaches still provide what are essentially reflective metaprogramming constructs, whereas our approach is located exclusively at the base level of the programming language and is based on the considerably simpler view that aspects are just local redefinitions of functions with dynamic extent.

Aspect-Oriented Programming has been characterized as a combination of quantification and obliviousness [5] (see also [4]). Obliviousness means that aspects can define certain, typically non-functional, properties of target code without the need for the target code to have an explicit awareness of those aspects. Dynamic scope for function definitions captures this obliviousness dimension of

AOP very well: code that uses certain functions can run in contexts with or without varying dynamically scoped redefinitions of those functions. The quantification dimension is not directly covered by dynamically scoped functions. Quantification means that an aspect definition can be applied to more than one place at a time, and is available for example in AspectJ by means of the pointcut construct that can span a multitude of join points in the target code. However in Lisp dialects, quantification is already adequately captured as one of the many uses of structural macros. As an illustration of this point we give a sketch of a `multidflet` macro:

```
(defmacro multidflet ((functions &body def)
                                    &body body)
  `(dflet ,(mapcar
            (lambda (function)
              `(,function (&rest args) ,@def))
            functions)
     ,@body))
```

Its effect is that the single function definiton `def` is used for the dynamically scoped redefinitions of the list of function symbols `functions`. The elaboration of possible usage scenarios is left as an exercise to the reader.

4 Conclusions

This paper is a slightly extended version of a submission accepted for the workshop on "object-oriented language engineering for the post-Java era" at ECOOP 2003.[2] Hopefully, it reveals the following points, among others:

- Programming languages that support a tendency to focus on static properties of programs can be a hindrance to see the forest from the trees. The aspect-oriented programming community devotes lots of energy into the provision of complex static means to reason about eventual dynamic properties of a program. Only the focus on a *strongly* dynamically-natured language construct reveals that we are in fact dealing with a very simple mechanism.
- This mechanism is that of dynamic scoping for functions. Only the addition of a structured way to call the previous definition of a function, via `call-next-function`, is needed to turn it into a full-fledged aspect-oriented approach.
- Appendix A gives a complete executable implementation of the ideas presented in this paper, but the paper still respects the page limit as imposed both by that workshop and by SIGPLAN Notices. This is only possible because a sufficiently complete and well-balanced language is used.

[2] http://prog.vub.ac.be/∼wdmeuter/PostJava/

Acknowledgements The author thanks the following people for inspiration, comments and fruitful discussions, in alphabetical order: Eli Barzilay, Tim Bradshaw, Thomas F. Burdick, Jeff Caldwell, Paul Foley, Erann Gat, Steven M. Haflich, Robert Hirschfeld, Kaz Kylheku, Ralf Lämmel, Barry Margolin, Joe Marshall, Wolfgang De Meuter, Mario S. Mommer, Pekka P. Pirinen, Kent M. Pitman, Alexander Schmolck, Nikodemus Siivola. Special thanks to Xanalys for allowing me to use the beta version of LispWorks for Macintosh to perform some additional tests.

References

[1] ANSI/INCITS X3.226-1994. *American National Standard for Information Systems - Programming Language - Common Lisp*, 1994. See [18] for an online version of this document.

[2] J. Baker and W. Hsieh. *Runtime Aspect Weaving through Metaprogramming*. In: *AOSD 2002* - Proceedings. ACM Press, 2002.

[3] T. Bradshaw. *Maintaining dynamic state*, 2001. http://www.tfeb.org/lisp/hax.html#DYNAMIC-STATE

[4] R. Filman. *What Is Aspect-Oriented Programming, Revisited*. Workshop on Advanced Separation of Concerns, ECOOP 2001, Budapest, Hungary.

[5] R. Filman and D. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*. Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, USA.

[6] M. Flatt. *PLT MzScheme: Language Manual*, 2002. http://download.plt-scheme.org/doc/

[7] D. Gillespie. *Common Lisp Extensions*. Included in distributions of GNU Emacs and XEmacs. http://www.gnu.org/software/emacs/emacs.html and http://www.xemacs.org/

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[9] P. Graham. *On Lisp*. Prentice-Hall, 1993. http://www.paulgraham.com/onlisp.html

[10] D. Hanson and T. Proebsting. *Dynamic Variables*. In: *PLDI 2001* - Proceedings. ACM Press, 2001.

[11] R. Hirschfeld. *AspectS - Aspect-Oriented Programming with Squeak*. In: *Objects, Components, Architectures, Services, and Applications for a Networked World*. Springer LNCS 2591, 2003.

[12] R. Kelsey, W. Clinger, J. Rees (eds.). *Revised5 Report on the Algorithmic Language Scheme*. Higher-Order and Symbolic Computation, Vol. 11, No. 1, September, 1998 and ACM SIGPLAN Notices, Vol. 33, No. 9, October, 1998.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. *An Overview of AspectJ*. In: *ECOOP 2001* - Proceedings. Springer LNCS 2027, 2001.

[14] R. Lämmel. *A Semantical Approach to Method Call Interception*. In: *AOSD 2002* - Proceedings. ACM Press, 2002.

[15] R. Lämmel. *Adding Superimposition To a Language Semantics*. In: *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*. Technical Report, Department of Computer Science, Iowa State University, 2003.

[16] J. Lewis, J. Launchbury, E. Meijer, M. Shields. *Implicit Parameters: Dynamic Scoping with Static Types*. In: *POPL 2000* - Proceedings. ACM Press, 2000.

[17] W. De Meuter. *Monads as a theoretical foundation for AOP*. In: International Workshop on Aspect-Oriented Programming at ECOOP, 1997.

[18] K. Pitman (ed.). *Common Lisp HyperSpec*, 2001. http://www.lispworks.com/reference/HyperSpec/

[19] G. Steele. *Common Lisp the Language, 2nd Edition*. Digital Press, 1990. http://www-2.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

[20] G. Steele and R. Gabriel. *The Evolution of Lisp*. In: T. Bergin, R. Gibson. *History of Programming Languages II*. Addison-Wesley, 1996.

[21] G. Steele and G. Sussman. *Lambda - The Ultimate Imperative*. MIT AI Lab, AI Lab Memo AIM-353, March 1976.

[22] G. Steele and G. Sussman. *The Revised Report on Scheme - A Dialect of Lisp*. MIT AI Lab, AI Lab Memo AIM-452, January 1978.

[23] G. Steele and G. Sussmann. *The Art of the Interpreter or, the Modularity Complex (Parts Zero, One, and Two)*. MIT AI Lab, AI Lab Memo AIM-453, May 1978.

[24] D. Tucker and S. Krishnamurthi. *Pointcuts and Advice in Higher-Order Languages*. In: *AOSD 2003* - Proceedings. ACM Press, 2003.

A The complete source code for `dflet`

```
 1: (defvar *dynsyms* (make-hash-table :test #'equal))
 2:
 3: (defmacro get-dynsym (fname) `(gethash ,fname *dynsyms*))
 4:
 5: (defun make-dynsym (fname)
 6:   (setf (get-dynsym fname) (make-symbol (format nil "*~A*" fname))))
 7:
 8: (defmacro ensure-dynsym (fname default)
 9:   (with-gensyms (sym) `(let ((,sym (get-dynsym ,fname)))
10:                          (if ,sym ,sym ,default))))
11:
12: (defun ensure-dynfun-form (fname &rest rest)
13:   (let ((dynsym (ensure-dynsym fname (make-dynsym fname))))
14:     `(progn (setf (get-dynsym ',fname) ',dynsym)
15:             (defparameter ,dynsym
16:               ,(if rest
17:                  `(lambda ,@rest)
18:                  `(if (fboundp ',fname)
19:                     (fdefinition ',fname)
20:                     (lambda (&rest args)
21:                       (cerror "Retry applying ~A to ~A."
22:                               "Undefined dynamic function ~A called with arguments ~A."
23:                               ',fname args)
24:                       (apply ',fname args)))))
25:             (defun ,fname (&rest args) (apply ,dynsym args)))))
26:
27: (defmacro defdynfun (fname args &body body)
28:   (apply #'ensure-dynfun-form fname args body))
29:
30: (defmacro redefdynfun (fname) (ensure-dynfun-form fname))
31:
32: (defun get-defined-dynsym (fname)
33:   (ensure-dynsym fname (progn (cerror "Make ~A a dynamic function."
34:                                       "Function ~A is not dynamic."
35:                                       fname)
36:                               (eval `(redefdynfun ,fname))
37:                               (get-dynsym fname))))
38:
39: (defmacro dflet1 ((fname (&rest args) &body funbody) &body dflbody)
40:   (let ((dynsym (get-defined-dynsym fname)))
41:     (with-gensyms (orgfun orgargs newargs)
42:       `(let* ((,orgfun ,dynsym)
43:               (,dynsym (lambda (&rest ,orgargs)
44:                          (flet ((call-next-function (&rest ,newargs)
45:                                   (apply ,orgfun (if ,newargs ,newargs ,orgargs))))
46:                            (destructuring-bind ,args ,orgargs ,@funbody)))))
47:          (declare (ignorable ,orgfun))
48:          ,@dflbody))))
49:
50: (defmacro dflet ((&rest decls) &body body)
51:   (reduce (lambda (decl result) `(dflet1 ,decl ,result)) decls
52:           :initial-value `(progn ,@body) :from-end t))
```