How to Make Lisp More Special

Pascal Costanza, Pascal.Costanza@vub.ac.be
Vrije Universiteit Brussel, Programming Technology Lab
Pleinlaan 2, B-1050 Brussels, Belgium

Abstract

Common Lisp provides generalized places that can be assigned via the SETF macro, and provides ways to hook into SETF. Past attempts to extend this to rebinding places, similar to "special" variables, lead to potentially incorrect behavior: New "bindings" are created by side effects, and therefore affect all threads that access the same places. Instead of storing values directly we can store symbols in those places and associate the values as symbol values. This allows new bindings to be confined to the current thread. As an illustration, I provide a DLETF framework and a CLOS metaclass SPECIAL-CLASS.

1 Introduction

Common Lisp provides a notion of generalized places that can be assigned with SETF. The SETF macro analyzes the form it is applied to and expands into a form that performs the actual assignment statement. Common Lisp provides various ways to hook into that SETF framework. One of the simplest ways is to just define a function with a name that is not a symbol as usual, but a list consisting of SETF and the name for which the SETF macro should expand into a call to that function. As an example, consider a getter and a setter function for a closed-over value:

```
(let (some-value)
  (defun some-value ()
    some-value)
  (defun (setf some-value) (new-value)
    (setq some-value new-value)))
```

Now, one can call (some-value) to retrieve the value of the lexical variable some-value, and (setf (some-value) 42) to set the value of that variable. The latter form may, for example, expand into (LET NIL (FUNCALL #'(SETF SOME-VALUE) 42)). The main advantage of the SETF framework is that it provides and supports a consistent naming scheme for getters and their associated setters. See [17] for the specification of the SETF framework.

There have been attempts in the past to provide a similar framework for rebinding places instead of assigning them. For example, McCLIM [14] defines a LETF form for temporarily binding slots to new values. In the following code snippet, for example, a 'medium' object is bound to specific values for 'ink' and for 'style'. These bindings are kept for the dynamic extent of the enclosed code.

```
(letf (((medium-ink medium)        +red+)
       ((medium-line-style medium) +bold+))
  (draw-line medium x1 y1 x2 y2))
```

The anticipated effect of such a construct is similar to the effect of rebinding a dynamically scoped "special" variable in Common Lisp. Special 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.

In the LETF example above, all forms that refer to the respective 'ink' and 'style' slots also see the new values in the dynamic extent of the LETF form. It is important to note that LETF does not introduce new access functions (MEDIUM-INK and MEDIUM-LINE-STYLE in the example above) that shadow the previously visible accessors, but that the change to slot values is indeed performed on the actual slots of the MEDIUM object. This means that if the MEDIUM is (or has been) passed around to other functions, the new slot values will be visible during the extent of the LETF form, no matter how they are accessed.

The LETF operator is usually implemented by expanding into an UNWIND-PROTECT form that stores the old values in temporary variables, changes the slots via SETF and changes them back to their old value in the protected forms. This indeed gives us the expected semantics for the dynamic extent of the LETF form. However, this also leads to potentially incorrect behavior in the presence of multi-threading since the changes to the slots are not confined to the current thread, but also affect all other threads that happen to access these slots in the respective objects. Effectively, this is closer to the more adequately termed LET-GLOBALLY construct of some Lisp dialects and libraries, for example ZetaLisp and Liquid Common Lisp, than to the envisioned rebinding of special variables. While LET-GLOBALLY and the sketched implementation of LETF change global bindings, new bindings of special variables in Common Lisp are always locally confined to the current thread in all multi-threaded Common Lisp implementations that I am aware of. In the *PRINT-BASE* example, say, the value 16 is only seen in the thread executing the code fragment above but not in others, unless by coincidence. The unwritten rule to confine new bindings of special variables this way is also reported in other publications, like [5].[1]

2 Dynamic Scope for the Masses

On a superficial level, it seems impossible to introduce correct dynamic scoping, as available for special variables, to other kinds of references in Common Lisp because storage locations are inherently "singletons" that cannot automagically "multiplex". However, Common Lisp specifies the PROGV special operator that provides a handle on dynamic scoping. I have combined this operator with the fact that all problems in computer science can be solved by another level of indirection, and implemented the DLETF framework. As explained in the following paragraphs, it combines two ideas: It uses PROGV for rebinding values with correct dynamic scoping, and it provides ways to hook into that framework to define one's own forms that work with DLETF, providing a similar level of extensibility as SETF.

2.1 PROGV

The PROGV operator is one of Common Lisp's special operators. It takes a list of symbols, a list of values and an implicit block of code, creates new dynamic bindings for the symbols with the given values, and executes the code block in that dynamic environment. Its explicit purpose is to provide a handle on the mechanism for binding special variables [17].

Although there is no guarantee that PROGV works as expected in multi-threaded implementations of Common Lisp, according to the expectations described above, PROGV also provides the behavior of confining new bindings to the current thread in all

[1] LETF has its roots in an operator of the same name on Lisp Machines. Reportedly, Lisp Machines provide machine-level operations that makes it straightforward to implement LETF with the correct semantics akin to special variables, so the problems described above did not exist on Lisp Machines.

implementations that I am aware of, following the consensus for special variables.

This results in the following scheme to provide correct dynamic scoping in a generic way: Instead of storing values directly in “special” places, (uninterned) symbols are stored in those places, and the values are associated as the respective symbol values. This allows DLETF to rebind those symbol values to new values, and given that PROGV is implemented as expected, this correctly confines those new bindings to the currently executing thread.

2.2 Hooking into DLETF

The DLETF framework, as I have defined it, does not provide any new kinds of special places by itself. However, any place / accessor can be used in conjunction with DLETF, provided they adhere to the following simple protocol: It is the responsibility of the accessors to store symbols instead of values in the respective places. They must use the operator MAKE-SPECIAL-SYMBOL of the DLETF framework to create such symbols in order to allow checking correct usage, and can use the SPECIAL-SYMBOL-P predicate to distinguished them from other symbols.

When the accessors are invoked, they have to check the setting of *SYMBOL-ACCESS*: When it is bound to a non-nil value, they have to write or read the special symbol as such instead of the actual symbol value, otherwise they have to access the symbol value. The special symbols stored in the special places can be created lazily on demand, but must remain the same after initialization.

As an example, we take a look at our introductory code snippet again.

```
(dletf (((medium-ink m)        +red+)
        ((medium-line-style m) +bold+))
  (draw-line m x1 y1 x2 y2))
```

The DLETF macro expands this code into the following application of PROGV.

```
(progv (let ((*symbol-access* t))
         (list (medium-ink m)
               (medium-line-style m)))
       (list +red+ +bold+)
  (draw-line m x1 y1 x2 y2))
```

That code switches *SYMBOL-ACCESS* to T to get the symbols associated with the respective slots, binds those symbols to the new values, and then executes the DRAW-LINE form.

2.3 The SPECIAL-CLASS Metaclass

In order to make the latter example work in full, I provide a CLOS metaclass SPECIAL-CLASS that allows declaring slots to be “special”, which means that their accessors adhere to the DLETF protocol. This allows us to declare the “medium” class as follows.

```
(defclass medium ()
  ((ink :accessor medium-ink
        :special t)
   (line-style :accessor medium-line-style
               :special t))
  (:metaclass special-class))
```

The metaclass SPECIAL-CLASS is implemented as follows: In the class initialization and class finalization stages,[2] slots are checked whether they are declared :SPECIAL, and (only) such special slots are then represented as instances of SPECIAL-EFFECTIVE-SLOT-DEFINITION. This allows the slot access functions to be defined according to the DLETF protocol. As an example, see the following definition of SLOT-VALUE-USING-CLASS.

```
(defmethod slot-value-using-class
  ((class special-class)
   object
   (slot special-effective-slot-definition))
  (let ((slot-symbol (call-next-method)))
    (cond (*symbol-access* slot-symbol)
          ((boundp slot-symbol)
           (symbol-value slot-symbol))
          (t (slot-unbound ...)))))
```

[2]According to the CLOS MOP specification given in [12], class initialization records the information for the given class only, without taking into account the specifications of the superclasses, while class finalization takes the final steps to allow objects of a class to be instantiated. This is a two-step process in order to ease development and automated build processes by allowing classes and their superclasses to be defined in arbitrary order.

The setter for SLOT-VALUE-USING-CLASS must invoke the getter to read the symbol associated with the respective slot:

```
(defmethod (setf slot-value-using-class)
  (new-value
   (class special-class)
   object
   (slot special-effective-slot-definition))
  (if *symbol-access*
    (call-next-method)
    (let ((slot-symbol
           (let ((*symbol-access* t))
             (slot-value-using-class
              class object slot))))
      (setf (symbol-value slot-symbol)
            new-value))))
```

This setter has to check for *SYMBOL-ACCESS* because being able to set the symbol is important for initializing special slots. The other generic functions for accessing slots (SLOT-BOUNDP-USING-CLASS and SLOT-MAKUNBOUND-USING-CLASS) are also implemented accordingly. Furthermore, slots that do not declare an :INITFORM option in the respective class definition are lazily associated with a symbol for DLETF. This is taken care of with a specialization of the SLOT-UNBOUND function.

Two other technicalities need to be solved.

- CLOS allows bypassing the slot access functions (SLOT-VALUE-USING-CLASS, etc.) for initializing freshly allocated objects for efficiency reasons. This is rectified in a method for SHARED-INITIALIZE that checks for special slots that are not associated with special symbols as generated by MAKE-SPECIAL-SYMBOL but with "ordinary" values, and "lifts" such values to special symbol values as required by DLETF.

- CLOS allows classes to be redefined at runtime. The SPECIAL-CLASS implementation has to guard against switching a special slot to a non-special slot in such a redefinition, because there is no obvious way to "collapse" the potentially many values of a single special slot in different threads to a single value of a non-special slot. The reverse change from a non-special to a special slot is not a problem and is implicitly dealt with by lifting the slot value in SHARED-INITIALIZE, as described above.

2.4 Other Data Structures

The CLOS Metaobject Protocol proves to be very suitable for adding special slots to the DLETF framework. Other data structures, like arrays, lists, structures, etc., do not provide metaobject protocols that allow changing their semantics in similar ways. In order to make their accessors adhere to the DLETF protocol, one has to revert to some other means. Common Lisp provides packages as a modularization mechanism, and they allow "shadowing" existing symbols of other packages. This indeed allows us to redefine existing accessors and make existing code "automagically" use the new definitions.

Assume we want to allow arrays to have special entries. We can define a package SPECIAL-ARRAY that imports all Common Lisp definitions but shadows MAKE-ARRAY and AREF, and redefines them as follows.

```
(defpackage "SPECIAL-ARRAY"
  (:use "COMMON-LISP" "DLETF")
  (:shadow "MAKE-ARRAY" "AREF")
  (:export "MAKE-ARRAY" "AREF"))

(in-package "SPECIAL-ARRAY")

(defun make-array (dimensions &rest args)
  (let ((array
         (apply #'cl:make-array
                dimensions args)))
    ... initialize with special symbols ...
    array))

(defun aref (array &rest subscripts)
  (let ((symbol
         (apply #'cl:aref array
                subscripts)))
    (if *symbol-access*
      symbol
      (symbol-value symbol))))
```

This code is clearly only a sketch. We may want to change the code to allow lazy initialization with special symbols in case an array is not initialized itself. We also have to ensure that other array (and sequence) accessors adhere to the `DLETF` as well. Finally, we may want to turn `MAKE-ARRAY`, `AREF`, etc., into generic functions that deal both with regular and special arrays by implementing methods for those different kinds of arrays, which means that we would effictively define an object-oriented layer for array accesses.

None of these issues are severe stumbling blocks. They are just tedious and therefore left as an exercise to the reader.

2.5 Efficiency

Carrying out considerably large benchmarks to test the performance overhead induced by the `DLETF` framework is still future work at the time of writing this paper. Therefore, I can only provide a few comments on the design decisions for the existing implementation.

Dynamic Scoping It is known that rebinding special variables induces a performance overhead. For example, [2] describes three implementation techniques called *shallow binding*, *deep binding* and *acquaintance vectors*, each with their own performance characteristics. Shallow binding associates one global storage location for a special variable and on rebinding, the old value is stashed away in a stack of previous bindings. This makes accessing and rebinding fast but context switches between different threads slow because all special variables have to be changed on each switch. Deep binding uses an association list as an environment for special variables that maps variable names to values. This makes rebinding and context switches fast but accessing variables slow because in the worst case, the whole association list has to be searched for the current binding of a special variable. Finally, acquaintance vectors use arrays as an environment for special variables, with each variable having a unique index into such arrays. This makes accessing and context switching fast but rebinding slow because for each new binding a whole array has to be copied. See [2] and also [10] and [1] for more details.

The slow aspects of these implementation techniques can (and reportedly are) improved by caches. In the (admittedly small) examples I have tried so far, I have not noticed serious penalties because of the use of rebinding symbol values.

Double Indirection The `DLETF` framework works by storing symbols into storage locations and associating the respective values with those slots instead of storing the values directly into the storage locations. This leads to the need to follow an extra indirection for each access to a special place. Whether this incurs a serious penalty or not can only be determined by performing benchmarks. Existing literature suggests that double indirection does not, or at least does not necessarily lead to any performance penalty at all on modern CPUs. See, for example, [16] and [8].

Slot Access Specializing the slot access functions (`SLOT-VALUE-USING-CLASS`, etc.) seems to incur a serious performance overhead for the seemingly innocuous task of accesing an object slot. It seems to be a serious penalty that these functions exist at all because one would expect a serious slowdown in comparison to the simple array accesses that other object-oriented languages usually compile slot accesses to. However, the CLOS MOP is designed in a way that allows a Common Lisp compiler to completely bypass the slot access functions when it can prove that for “simple” slots there does not exist a specialization of the respective slot access function. This is why in my implementation of `SPECIAL-CLASS`, only slots with `:SPECIAL` set to `T` are represented by instances of `SPECIAL-EFFECTIVE-SLOT-DEFINITION`. Examination of the resulting code of a number of CLOS implementations indeed reveal that they make use of the possible optimizations for non-special slots. So at least, the performance overhead is confined to slots that are indeed special.

In [15], an improvement of the CLOS slot access protocol (“structure instance protocol”) is suggested that may help to reduce the overhead of invoking the code of the special slot accessors. This approach is implemented, for example, in Tiny CLOS for Scheme and its derivatives. It is probably worthwhile to implement the special classes on top of such an approach and compare the performance characteristics with my CLOS-based implementation.

3 Related Work

Scheme does not provide any standard constructs for dynamic scoping - in [18], 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 MzScheme [9], provide dynamic scope in two ways: as the `fluid-let` form and as parameter objects. 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, similar to `LET-GLOBALLY` or McCLIM's `LETF`, and so has the same issues with multi-threading that `DLETF` solves. On the other hand, parameter objects and its `parameterize` form are indeed viable candidates to be used in the same way as symbols and `PROGV` in `DLETF`.

Recent attempts at introducing dynamically scoped variables into C++ [11] and Haskell [13] would in principle also be suitable because they are implemented by passing dynamic environments to functions behind the scenes instead of modifying global state. However, it is not clear whether those approaches provide first-class handles on their dynamic binding constructs as `PROGV` or parameter objects do.

4 Summary and Future Work

The `DLETF` framework is currently available as part of AspectL, a library for aspect-oriented programming in Common Lisp, and can be downloaded from http://common-lisp.net/project/aspectl. It runs on Allegro Common Lisp, CMU Common Lisp, LispWorks, OpenMCL and Steel Bank Common Lisp.

One of the main problems with implementing AspectL, including the metaclasses `SPECIAL-CLASS` and `SPECIAL-FUNCTION`, were the incompatibilities of the CLOS MOP implementations across different Common Lisps. Since AspectL has started to grow a considerable amount of fixes to iron out those incompatibilities, I have started the implementation from scratch and now provide two separate packages: A "Closer to MOP" library that provides a unified interface to the CLOS MOP with many rectifications of those incompatibilities, and ContextL. Closer to MOP can be downloaded from http://common-lisp.net/project/closer and runs on CLISP (2.33) in addition to the implementations mentioned above. On top of that, ContextL includes a somewhat polished version of the `DLETF` framework and a new implementation of `SPECIAL-CLASS`, among other things. ContextL is going to supersede AspectL, and this paper describes the implementation of `DLETF` and `SPECIAL-CLASS` in ContextL, not the one in AspectL. Unfortunately, ContextL is not publicly available yet at the time of writing this paper.

AspectL's `SPECIAL-FUNCTION` is an interesting application of the `DLETF` framework. Conceptually, it turns the slot of a generic function metaobject that contains the methods defined on a generic function into a special slot. This means that one can rebind the set of generic function methods in the same way as special variables, and so one can have different sets of methods in different threads for the same generic function. This in turn allows expressing interesting idioms that are all based on the idea to change the behavior of a function with dynamic extent, and thus are similar to dynamically scoped functions. For example, functions can be wrapped with security checks in some threads, or expensive methods can be replaced by stubs in testing frameworks, and so on. Special functions are, in fact, an implementation of the originally envisioned `WITH-ADDED-METHODS` macro, as specified in the original CLOS specification [3], but with dynamic scope instead of lexical scope.[3] See [6]

[3]`WITH-ADDED-METHODS` was not included in ANSI Common

for an introduction to dynamically scoped functions and [7] for an overview of AspectL, including `SPECIAL-FUNCTION`.

The use of symbols to "multiplex" arbitrary places is, in a sense, the reverse of the use of symbols in Common Lisp's synonym streams to make virtually different streams in fact refer to the same single stream. Rebinding the symbol of a synonym stream rebinds all existing synonyms for that stream. I have not explored yet what a combination of that approach with `DLETF` could yield in terms of expressivity. The idea would be to assign the same special symbol to different places and in this way have them permanently linked to each other.

References

[1] Henry Baker. Shallow Binding in Lisp 1.5. Communications of the ACM 21, 7 (July 1978), 565-569. Available: http://home.pipeline.com/~hbaker1/ShallowBinding.html

[2] Henry Baker. Shallow Binding Makes Functional Arrays Fast. ACM Sigplan Notices 26, 8 (Aug. 1991), 145-147. Available: http://home.pipeline.com/~hbaker1/ShallowArrays.html

[3] Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, David Moon. Common Lisp Object System Specification. Lisp and Symbolic Computation 1, 3-4 (January 1989), 245-394. Available: http://www.dreamsongs.com/CLOS.html

[4] CLOS Standardization Mailing List (archive). Available: ftp://ftp.parc.xerox.com /pub/pcl/archive

[5] Roger Corman. Multi-Threaded Lisp: Challenges and Solutions. International Lisp Conference 2002, San Francisco, USA, 2002.

Lisp due to being underspecified. The archive of the CLOS standardization mailing list [4] suggests that it may have originally been supposed to be dynamically scoped, but this is not completely clear.

[6] Pascal Costanza. Dynamically Scoped Functions as the Essence of AOP. ECOOP 2003 Workshop on Object-Oriented Language Engineering for the Post-Java Era, Darmstadt, Germany, July 22, 2003. ACM Sigplan Notices 38, 8 (August 2003). Available: http://www.pascalcostanza.de/dynfun.pdf

[7] Pascal Costanza. A Short Overview of AspectL. European Interactive Workshop on Aspects in Software (EIWAS'04), Berlin, Germany, September 23-24. Available: http://www.topprax.de/EIWAS04/

[8] Pascal Costanza. *Transmigration of Object Identity.* University of Bonn, Institute of Computer Science III, Ph.D. thesis, 2004.

[9] Matthew Flatt. *PLT MzScheme: Language Manual*, 2005. Available: http://download.pltscheme.org/doc/

[10] Richard Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985. Available: http://www.dreamsongs.com/Books.html

[11] David Hanson and Todd Proebsting. Dynamic Variables. In: *PLDI 2001* - Proceedings. ACM Press, 2001.

[12] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[13] Jeffrey Lewis, John Launchbury, Erik Meijer, M. Shields. Implicit Parameters: Dynamic Scoping with Static Types. In: *POPL 2000* - Proceedings. ACM Press, 2000.

[14] McCLIM. Available: http://common-lisp.net /project/mcclim/

[15] Fernando D. Mato Mira. The ECLOS Metaclass Library. In: Chris Zimmermann (ed.). *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 2000.

[16] Sven Müller. *Transmigration von Objektidentitäten – Integration der Spracherweiterung*

Gilgul in eine Java-Laufzeitumgebung. University of Bonn, Institute of Computer Science III, diploma thesis, 2002.

[17] Kent Pitman (ed.). Common Lisp HyperSpec. Available: http://www.lispworks.com /documentation/HyperSpec/

[18] Guy Steele and Gerald Sussman. *Lambda - The Ultimate Imperative*. MIT AI Lab, AI Lab Memo AIM-353, March 1976. Available: http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-353.pdf