A Short Overview of AspectL

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

August 21, 2004

1 Introduction

AspectL adds a number of features to the Common Lisp Object System (CLOS) that have been developed in the recent years in the AOSD community. According to the Lisp spirit, some of them have been generalized in the process of translating them to Common Lisp.

These generalizations are likely to be interesting for designers of other aspect-oriented programming languages as well since the bulk of AspectL does not rely on Common Lisp specifics. However, this paper requires at least a general understanding of CLOS, as for example provided in the overviews by DeMichiel and Gabriel [2] and Gabriel, White and Bobrow [5]. Especially the notion of *generic functions* that are collections of methods, and the fact that it is the generic functions and not the methods that are called should be well understood.[1] An understanding of the CLOS Metaobject Protocol (MOP) is not required - it is only mentioned in notes that are not essential to the rest of the presentation.[2]

2 Generic Pointcuts

Generic pointcuts are collections of join points and aspect weavers. A join point can be, more or less, any description of some event in the control flow of a (CLOS) program, but typically it denotes the call of a method. You can define a join point together with some arguments that are subsequently passed to aspect weavers that are in turn applied to all the join points in a pointcut. An aspect weaver is a function that must install and return a method, given a specific join point.

[1] The term "generic function" refers to the fact that it is a collection of methods, and that usually only a subset of those methods is applicable for some concrete arguments. It is the generic function's responsibility to sort out the correct selection and combination of those methods according to the classes or identities of the arguments. Note that in this context, "generic" is not related to notions of generic types of statically typed languages, or other arbitrary uses of that word.

[2] However, see Paepcke's excellent introduction [10] if you are still interested in the CLOS MOP, and the book "The Art of the Metaobject Protocol" by Kiczales, des Rivières and Bobrow [7] for the whole story.

Here is an example: If you want to add bounds checking to classes that represent graphical objects, you can do the following. Assume that, for example, a class `point` is defined as follows.

```
(defclass point ()
  ((x :accessor x :initarg :x :type integer)
   (y :accessor y :initarg :y :type integer)))
```

The `:accessor` declarations in CLOS make sure that appropriate getter and setter methods are automatically generated. Assume that one wants to make sure that the respective slots do not exceed certain bounds. In general, this can already be achieved in CLOS by way of before/after/around methods, as follows.[3]

```
(defconstant min-x -20)
(defconstant max-x 20)

(defmethod (setf x) :before (new-value)
  "Check the bounds for x's setter."
  (unless (<= min-x new-value max-x)
    (error "x is out of bounds.")))
```

However, a more declarative way to specify bounds is desirable, and this can be achieved via generic pointcuts in AspectL. First, we define a generic pointcut that allows for addition of aspect weavers and join points. Note that in AspectL, a generic pointcut does not define any of those of its own but only declares a name for the pointcut that aspect weavers and join points can refer to later on.

```
(define-pointcut bounds-checking)
```

Next, we define an aspect weaver that processes join points. The aspect weaver declares a function that accepts two metaobjects that represent the aspect weaver which defined it and the join point to which it is applied, and optionally a number of additional arguments as needed by the concrete aspect weaver. In this case, the aspect weaver and join point metaobjects are not needed for creating the method that performs the bounds checks. Note that we also have to provide a name for the aspect weaver (`check-coordinate`), but it is only required for supporting interactive program development.[4]

```
(define-aspect-weaver bounds-checking check-coordinate
    (aspect-weaver join-point set-function slot-name min max)
  (declare (ignore aspect-weaver join-point))
  (create-method
    set-function
    :qualifiers '(:before)
    :lambda-list '(new-value)
```

[3] Such before/after/around methods are not part of AspectL but are already part of CLOS since its conception in the 1980's. In fact, they have been influenced in turn by the notion of advice functions in the Interlisp-family of dialects in the early 1970's. See Teitelman [12, 13].

[4] This simplifies replacing existing aspect weavers with new definitions in a similar way as CLOS already does in the case of methods which are automatically replaced by new methods whose arguments have the same specializers.

```
  :specializers (list (find-class 't))
  :body `(unless (<= ,min new-value ,max)
           (error "~S is out of bounds" ',slot-name))))
```

(The `create-method` function provides a simplified way to create new methods programmatically, and just performs appropriate calls to `make-method-lambda` and `add-method`, as defined by the CLOS MOP [7].)

Now, it is possible to declare join points for that pointcut that are, as a result, purely declarative means to define the bounds for coordinates of graphical objects.

```
(define-join-point bounds-checking set-x
  (function (setf x)) 'x -20 20)

(define-join-point bounds-checking set-y
  (function (setf y)) 'y -40 40)
```

Note that we have to provide a name for the join point, again in order to support interactive program development.[5]

Generic pointcuts in AspectL are unique in that they textually separate the definition of what is widely known as advice code and the actual join points, which are usually encapsulated in the same aspect declaration. This allows the programmer to arbitrarily place join point and aspect weaver definitions that refer to the same pointcut anywhere in a program, in arbitrary order. In fact, the `define-pointcut` form does not need to be mentioned at all but is implicitly created by the first aspect weaver or join point definition that mentions a new pointcut name. This is similar to the way how generic functions are implicitly defined by method definitions in CLOS – and this is why they are accordingly called generic pointcuts. An important advantage of the design of generic pointcuts is that join point definitions can appear either near the aspect weavers or near the entities (slots, methods) that they refer to, which effectively means that they are a unification of aspect-oriented and metadata annotation approaches, like attributes in C# [3] or the metadata facility in Java's forthcoming JDK 5.0 [sic!] [11].

Generic pointcuts provide a way to add methods to a CLOS program that deal with crosscutting concerns. The base program is oblivious to those added method definitions, and the aspect weaver definitions allow expressing method definitions in a single place that ultimately expand to a number of corresponding actual methods. There is no direct support for quantified expressions that declaratively yield a number of actual join points. Instead, meta-level facilities are provided to add, remove or change join points and aspect weavers, which opens up the possibility of using any of the various collection and/or comprehension libraries that exist for Common Lisp. What is still missing though at this point in the presentation are a facility to add, remove or change slots ("fields")

[5] Furthermore, one can notice that there is apparently an unnecessary redundancy with regard to how often the name of the slot needs to be repeated in the join point declarations. This can, in fact, be eliminated within the aspect weaver definition by way of the introspection facilities of the CLOS MOP: The set function could be asked what slot it affects, or the other way around, and the join point name could actually be the name of the set function and/or the slot. This can potentially lead to an elimination of two parameters in the join point declarations, but has been left out for the sake of clarity.

in classes on the one hand, and a way to reason about the control flow of a program on the other hand. These tasks are handled by *destructive mixins* and *special generic functions* respectively, and are described in the remainder of this paper.

3 Destructive Mixins

A "destructive" mixin is a function that incrementally modifies the definition of an already existing class. For example, assume you have a class `person` defined as follows.

```
(defclass person ()
  ((name :accessor name :initarg :name)))
```

At a later stage in the program, you can add new slots to such a class, without repeating the complete class definition, as follows.

```
(with-class 'person
  (class-add
   :direct-slots
   '(age :accessor age :initarg :age)))
```

This can be considered as an aspect-oriented extension of CLOS because the two forms that define aspects of the person class can indeed be defined separately. This is similar to the notion of contextual class extension [14] and is, in fact, modeled after the notion of destructive mixins as described in [15]. Destructive mixins in AspectL can add, remove or change class options for a class by way of `class-add`, `class-remove` and `class-set`, and add, remove or change slot options by way of `slot-add`, `slot-remove` and `slot-set`. Effectively, all the aspects of class definitions that can be affected by way of the CLOS MOP can also be affected via destructive mixins.

The implementation of destructive mixins is a relatively straightforward combination of the introspection capabilities of the CLOS MOP and its `ensure-class` function that allows redefinition of existing classes. Since CLOS also already handles updating of existing instances in a controlled way, the only contribution of AspectL's destructive mixins is indeed a reconstruction of a class definition via introspection and the subsequent merging with changes requested from a client program.

4 Generalized Special Places

Common Lisp provides a notion of generalized places that can be assigned with `setf`. The `setf` framework analyzes the form it is applied to and expands into an actual assignment statement. We have seen an example in the section about generic pointcuts above: `(setf x)` and `(setf y)` are the setters to the (dual) `x` and `y` getter functions for the respective slots, and they can be called via the `setf` macro, for example as in `(setf (x p) 5)`, given that `p` is an instance of the class `point` above.

In the Common Lisp community, there have been attempts in the past to provide similar frameworks for rebinding places instead of assigning them. This

can be used for example in GUI libraries to temporarily set fields of some object to certain values and let them automatically be set back to their original state. See the following code.

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

Here, a `letf` macro is used to bind the ink and the style of a graphical output medium to specific values, draw a line, and then to revert to the medium's previous state.[6] This is a behavior that would otherwise probably be simulated by way of storing the old values in temporary local variables, assigning the new values, and finally storing the old values back into their original places in appropriate `unwind-protect` forms (also known as `try-finally` blocks in other languages). In fact, this ad hoc idiom is usually used in `letf` implementations which use the corresponding `setf` functions internally. However, the problem with that approach is that such a temporary assignment to an object's field is globally visible, potentially affects other threads, and therefore may lead to incorrect behavior in multi-threaded scenarios.

AspectL provides a framework for a correct handling of such generalized rebindings. In order to stress that dynamic scoping is respected, AspectL uses the name `dletf` as the binding operator instead of that traditional `letf`. Instead of reverting to `setf`, `dletf` makes use of the fact that symbol values can be rebound with dynamic extent via the standard operator `progv`, and relies on the Common Lisp implementation to implement `progv` correctly with regard to multi-threading. In turn, `progv` already provides the necessary machinery for dynamically scoped variables, which are traditionally called "special variables".[7] AspectL's `dletf` framework only provides the framework for such generalized "special places". The actual special places that are to be specially rebindable need to be provided by other libraries, but need only adhere to a simple protocol for distinguishing between accesses to the special symbols and their symbol values. However, AspectL fortunately also provides special classes and special generic functions that provide useful applications of the `dletf` framework, and they are described in the following sections.

4.1 Special Classes

Special classes allow for declaration of special slots that can be dynamically rebound with `dletf`. Only the slots of a class that are declared to be "special" can be dynamically rebound, and this special declaration can only be used in classes whose metaclasses are `special-class`.[8]

[6] This example is due to Arthur Lemmens.

[7] See [1, 6, 8] for good introductions to special variables, and [9] for a formal account.

[8] The name "special class" may not seem very instructive to non-Lispers, but since the term "special variable" is very common and well-understood in the Common Lisp community, it is in fact the most descriptive one can choose. Alternatives for the same concept could have been *fluid class* in Scheme and *dynamic class* in ISLISP. (Likewise, special generic functions (see below) could have been called *fluid generic functions* or *dynamic generic functions*, respectively.)

Here is an example:

```
(defclass person ()
  ((name :accessor person-name :initarg :name :special t))
  (:metaclass special-class))

(defvar *p* (make-instance 'person :name "Dr. Jekyll"))

(dletf (((person-name *p*) "Mr. Hide"))
  (print (person-name *p*)))
=> "Mr. Hide"

(person-name *p*)
=> "Dr. Jekyll"
```

We can see that the `dletf` form temporarily assigns a new value to a special slot but that after return from the `dletf` form, the slot has its old value again. What we cannot see here is that the `dletf` form not only encapsulates the new value within its control flow, but also within the current thread. This means that other threads do not see the new value but still see the old value for the given slot. This effectively also means that the slot has two (or more) values at the same time.

4.2 Special Generic Functions

Special generic functions are like standard generic functions in CLOS, but additionally allow for adding methods with dynamic extent. This means that a special method can be declared to be executed only in the current dynamic scope, but not in others, and it is removed on exit from the dynamic scope of its definition. Effectively, this means that special generic functions do for methods what special classes do for slots. They are, in fact, a generalization of dynamically scoped functions, as I have described them in [1], in the sense that one can now also define before/after/around methods, possibly specialized on specific classes, with dynamic extent.

Here is an example: We provide the following special generic function for printing instances of the class `person` as defined above. It is a special generic function because we would like to customize its behavior later on according to the context in which it is used. The method definitions for special generic functions declare a first parameter that specifies for what scope the method definition is supposed to be active - this is either `t` for the global scope, or `dynamic` for the current dynamic scope. (It is idiomatic to use `t` for such purposes in Common Lisp which also denotes the boolean truth value, the standard output stream, the most general type, and other "general things".) In global definitions, we are only concerned with global scope:

```
(define-special-function print-person (person)
  (:definer print-person*)
  (:method ((scope t) (person person))
    (print (person-name person))))
```

Note that AspectL's special generic functions require two defining names instead of one as is the case for standard generic (and also ordinary) functions

in Common Lisp. The name in the defining form (`print-person` in the example above) can be used for calling a special generic function, whereas the name provided in the `:definer` option (`print-person*`) must be used for defining additional methods which require the additional scope parameter in their argument lists.[9]

Now, we provide another class `family` that declares relationships between persons that are closely akin to each other.

```
(defclass family ()
  ((mother :accessor family-mother :initarg :mother)
   (father :accessor family-father :initarg :father)
   (children :accessor family-children :initarg :children)))
```

The `print-family` function is complex because for each child, we want to print additional information about what family they belong to. So in the context of `print-family`, `print-person` needs to be redefined. This can be done in two steps: `with-special-function-scope` "opens" a new dynamic scope for the special generic functions provided as a parameter and within that new scope, new methods can be defined that are declared to be active for that dynamic scope. (The `dletf` operator cannot directly be used here because there is a need to set up additional internal auxiliary variables.) Here is the `print-family` function:

```
(defgeneric print-family (family)
  (:method ((family family))
   (print-person (mother family))
   (print-person (father family))
   (with-special-function-scope (print-person)
     (defmethod print-person* :after
         ((scope dynamic) (person person))
       (format t " mother ~S, father ~S"
               (mother family) (father family)))
     (loop for child in children
           do (print-person child)))))
```

This is a good example for a clean separation of concerns because the original `print-person` definition does not need to reason about the `family` class at all.

The requirement to use `define-special-function` instead of `defgeneric` may seem as a violation of obliviousness [4]. However, except for preparing these generic functions to be amended by methods with dynamic scope, the use of `define-special-function` does not really reveal anything interesting about its actual role in the program, so this is really only a very mild violation, required only to build the necessary infrastructure for the subsequent "real thing".

[9] This requirement stems only from the fact that Common Lisp compilers check defining forms and calling forms to have matching argument lists. In fact, previous versions of AspectL did not have this requirement because they were developed on a single Common Lisp implementation that is somewhat more liberal in this regard.

5 Conclusions and Future Work

The building blocks of AspectL that provide established functionality from other aspect-oriented approaches are:

- Generic pointcuts for static aspects on generic functions / methods.
- Destructive mixins for static aspects on classes / slots.
- Special classes for dynamic aspects on classes / slots.
- Special functions for dynamic aspects on generic functions / methods.

The separation into static and dynamic aspects refers to the fact whether the whole (static) program, independent of control flows or threads, or the current dynamic scope within the control flow of a program is affected. As stated above, special generic functions are a generalization of dynamically scoped functions as I have described them in [1]. While I am still convinced that the notion of dynamically scoped functions is the essential new feature of aspect-oriented programming beyond previous programming models and therefore, that paper provides a complete but only minimal account of AOP, AspectL now turns this idea into a more usable extension of an industrial-strength programming language. However, note that even the static features of AspectL provide means to be activated and deactivated again at runtime from a global scope, which essentially turn them again into means to express dynamically scoped abstractions, making it only more convenient to (de)activate them permanently.

As an overview, this paper can only deliver rough ideas how the functionality offered by AspectL can be used in actual programs. Internally, AspectL already makes use of the functionality provided in lower layers (generic pointcuts and `dletf`) in the implementation of the higher layers (special classes and special generic functions). Some aspects of AspectL's design are not completely fixed yet and may change in the future, according to the experiences made in real applications. The current version of AspectL is 0.6.1, at the time of writing this paper, and can be downloaded from http://www.common-lisp.net/project/aspectl/ under the Creative Commons Attribution License. It runs on Allegro Common Lisp (tested under 6.2 Trial Edition), CMU CL (tested on an experimental port to Mac OS X), LispWorks (tested under Personal Edition 4.3r2 for Mac OS X), OpenMCL (with restrictions, tested under 0.14.2-p1, Darwin port), and SBCL (tested under 0.8.13). Because of having ported it to five major Common Lisp implementations after its initial release for LispWorks, the code has stabilized considerably and should be easy to port to any ANSI-compliant Common Lisp implementation with a MOP that conforms to the specification given in [7].

Performance-wise, AspectL should not imply a considerable overhead because it largely makes use of standard functionality of CLOS and its MOP. So if the available CLOS + MOP implementation is fast, AspectL should be fast as well. A critical part may be the addition and removal of methods to special generic functions in a `with-special-function-scope` form, because they invalidate the caches that generic functions usually maintain for method selection and combination. As far as I can see at the moment, this could only be avoided by modifying the MOP implementation itself. However, benchmarks of large

applications of AspectL are not available yet, so this is all pure speculation anyway.

Future work includes more comprehensive descriptions of the use of AspectL features, especially in conjunction. They are likely to turn out especially useful for scenarios in which software needs to change its behavior according to the context in which it is used, leading to a new notion of *context-oriented programming* (and basically reiterating on the idea of dynamic scoping for functions). Work is underway with a number of research and industrial partners to explore this notion and its usefulness in a number of case studies.

Acknowledgements

I am thankful to the following people who have contributed, directly or indirectly, important insights and fruitful ideas that have considerably improved AspectL and its implementation: Frode Vatvedt Fjeld, Kaz Kylheku, Barry Margolin, Christophe Rhodes, Kevin M. Rosenberg, and Dan Schmidt. Further thanks go to the anonymous reviewers of the EIWAS '04 workshop for valuable suggestions for improvement of this paper.

References

[1] 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, published in ACM SIGPLAN Notices Volume 38, Issue 8 (August 2003), ACM Press. Available at http://www.pascalcostanza.de

[2] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An Overview. In: *ECOOP '87 - European Conference on Object-Oriented Programming*, Proceedings, Springer LNCS 276, 1987. Available at http://www.dreamsongs.com/CLOS.html

[3] ECMA International. *C# Language Specification*. Standard ECMA-334, 2nd Edition – December 2002. Available at http://www.ecma-international.org/

[4] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis.

[5] Richard P. Gabriel, Jon L. White, Daniel G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. In: *Communications of the ACM, Special Issue on Lisp*, 34(9):29-38, ACM Press, September 1991. Available at http://www.dreamsongs.com/CLOS.html

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

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

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

[9] Luc Moreau. A Syntactic Theory of Dynamic Binding. In: *Higher-Order and Symbolic Computation*, 11(3):233-279, December 1998.

[10] Andreas Paepcke. User-Level Language Crafting – Introducing the CLOS Metaobject Protocol. In: Andreas Paepcke (ed.), *Object-Oriented Programming: The CLOS Perspective*, MIT Press, 1993. Available at http://www-db.stanford.edu/∼paepcke/shared-documents/mopintro.ps

[11] Sun Microsystems, Inc. A Program Annotation Facility for the Java Programming Language. JSR-175 Public Draft Specification. Available at http://www.jcp.org/en/jsr/detail?id=175

[12] Warren Teitelman. Advising. Section 19 in: Warren Teitelman, Daniel G. Bobrow, Alice K. Hartley, Daniel L. Murphy, *BBN Lisp – Tenex Reference Manual*, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, USA, July 1971 (2nd Revision – August 1972). Available at http://bitsavers.org/pdf/bbn/tenex/

[13] Warren Teitelman. Advising. Section 19 in: Warren Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1974. Available at http://bitsavers.org/pdf/xerox/interlisp/

[14] Kresten Krab Thorup. Contextual Class Extensions. Aspect-Oriented Programming Workshop, ECOOP '97, Jyväskylä, Finland. Available at http://trese.cs.utwente.nl/aop-ecoop97/

[15] Kris De Volder. Inheritance with Destructive Mixins for Better Separation of Conerns. Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, Minnesota, USA. Available at http://trese.cs.utwente.nl/Workshops/OOPSLA2000/