CASL Single Lecture Notes

SLN - Lisp - 07

2005-10-01

Cognitive Autonomous Systems Laboratory
Department of Computer and Information Science
Linköping university, Sweden

Erik Sandewall

Approaches to Data Abstraction in Lisp

Subject Area: System Design in Lisp

Date of lecture: 2005-10-03

"Single Lecture Notes" are notes corresponding to one or a few lectures.

Related information can be obtained via the following WWW pages:

Linköping university:	http://www.liu.se/
This course:	http://www.ida.liu.se/~TDDA23/
CASL Group:	http://www.ida.liu.se/ext/casl/
The author:	http://www.ida.liu.se/~erisa/

1 Topic

The concept of *abstraction* is fundamental in the design of conventional programming languages. Important constructs in languages such as C++ and Java have been chosen in order to provide data abstraction at the core of the language. Lisp does not have that character – the language as such does not have any similar facilities.

This does not mean that data abstraction is considered as unimportant by Lisp users, but merely that Lisp is open to several different approaches to it. The following are some of the views that are represented in actual Lisp systems:

- The use of an extension of Lisp that provides object orientation, in particular the CLOS system (CommonLisp Object System). CLOS is defined as an IEEE standard, along with the standards definition of CommonLisp as such. CLOS is a large and relatively complex language, and will not be considered in this course.
- The use of macro-like facilities that can be defined by an individual user or a small user community, providing a simple extension of the Lisp language.
- The use of specialized languages that are embedded in Lisp, which means that Lisp is used as a high-level implementation language for the new language and as an execution environment for it. Specialized languages tend to provide not only data abstraction, but also control abstraction, since some of the control flow may be defined in the interpreter for the specialized language and not in the programs written by the user.

The present lecture note addresses the general issue of data abstraction and the second one of these three approaches, that is, the use of a simple extension of the language based on macro expansion. The use of specialized languages has been addressed in another lecture note.

There is also a possible argument that data abstraction is not as universally applicable in Lisp as in conventional languages, due to the use of a standardized, textual representation of data in Lisp, viz. the S-expressions. This argument will be addressed in the concluding discussion in this lecture note.

2 The what and why of data abstraction

The basic idea in data abstraction is twofold: programs should be written in a way that expresses concepts and structures in the application as understood by the programmer, and aspects of the program that are only motivated by implementation needs should be isolated to a section of their own that is kept as small as possible. This recommendation is motivated by several needs:

- *Ease of working with the program during its initial development phase:* making it easy for the programmer to understand her own or his own program, so that the number of errors is reduced, errors can be found as quickly as possible, and even the time of initial program design is minimized.

- *Ease of working with the program at a later time:* making it easy for the programmer to come back to the program for the purpose of making additions or changes.
- *Ease for others to work with the program.* This is similar to the previous item, but refers to persons that did not write the original program but need to change it later on.
- *Facilitate change of representation:* if one wishes to retain a program for a given application but change the way data are represented in the program, then it is convenient if the detailed data representation have been isolated, and on the other hand it is inconvenient if detailed data designs are reflected in many places in the code. Simply speaking, it is better if one does not have to change so many places in the code.
- *Provide the basis for checking against errors:* many trivial programming errors can be caught by automatic means if the program is written in a disciplined way using data abstraction.

All of these are important considerations. However, the present author would like to caution against fundamentalist positions where these are taken as absolute truths that one is not even allowed to question. Working with data abstraction imposes a certain overhead when the initial program is written. This investment in time is well spent if the program is going to be used for a long time, and by several developers, but there are also cases where one uses “throw-away” programs that are only used for a short time, so that the maintainability arguments bear less weight. Being able to change data representation is fine, but many revisions of program structure concern control flow and not merely the layout of data, and then it may turn out that the initially chosen data abstraction does not help at all. Everyone should understand the concepts of data abstraction, but like anything else they have to be applied with common sense.

The fact that data abstraction serves several goals at once also means that when you compare different approaches to the topic, you consider which of those goals apply in a particular context. For example, an approach that helps with ease of working with a program but does not help with automatic checking against errors, may still be useful in some situations and much less useful in others.

3 A concrete example: the state update functions in Ethel

The present lecture notes concern data abstraction in Lisp. We shall use a concrete example, namely, the definitions of state update functions in the Ethel demo application, which has been introduced in another lecture note. The Ethel base system contains several alternative definitions of the state update function for the action verb `let-board`, including a definition that does not use any data abstraction at all and that looks as follows:

```
(defun update-state1 ()
  (setq *t (+ 1 *t))
  (dolist (a *oa)
      ;; example: a is (action 5 (start-boarding lh-12))
      ;;  where lh-12 is airplane; 5 was the starting time
    (case (car (caddr a))
      (let-board
        (let ((x (assoc (cadr e) *ap)))
          (case (caadr x)
             (at-airport
                (rplaca (cdr x)
                        (list 'boarding (cadadr x) 1) ))
             (boarding
                (let ((j (cddr (cadr x))))
                     (cond ((equal (car j) 3)
                            (remaction e)
                            (rplaca (cdr x)
                                (list 'ready-to-fly
                                      (cadadr x) )))
                           (t (rplaca j (+ 1 (car j)))) )))
             (t nil) ))))

         ;; Additional action verbs are implemented by adding more
         ;; branches here for the outer case-expression

      (t (princ "Unknown action")(terpri))
              ))
  (logstate)
         )
```

This definition is written almost entirely in terms of elementary Lisp functions: `car`, `cdr` and their compositions, `rplaca`, `list`, `equal`, `setq`, and the control operators (`cond`, `case`, `let`). The only additional functions are the association-list lookup function, `assoc`, and the function `remaction` that is part of Ethel's action management system.

This way of writing the definition obviously violates the criteria that were mentioned above. The composite `car/cdr` expressions are difficult to read, in particular if you are not used to them. If one should change one's mind about how actions, action expressions, and the locations of persons and airplanes are to be represented, then one has to change those access expressions in many places. Consider, therefore, what are the possibilities of improving the situation within the general framework and philosophy of the Lisp language and system?

The use of a rule language. One possibility has been presented in the Ethel memo: introduce a rule language for situation-effect rules, implement an interpreter for that language, and write the state update functions for each action verb in terms of those rules, instead of directly as executable program code. This method is very effective when it works, but there are two objections to it: (1) it may be too restrictive with respect to what state changes can actually be expressed, and (2) at least as implemented in the Ethel memo, it assumes that actions and states are written in a particular way. If one should wish to change that representation, then again one has a problem of rewriting a lot of action rules.

The answer to the first objection will be that one has to use that technique judiciously, and be wise about when it is applicable and when it is not. The answer to the second objection may be that it is up to the system designer to design the rules so that they are sufficiently flexible with respect to possible future changes of representation, that this is no different from what is required from the designer of a data abstraction of a conventional kind, and that a change of representation for the expressions that are part of the rules can (maybe) be handled by a small program that rewrites the rules automatically.

The use of auxiliary functions Another possibility was illustrated in 'version 2' of the state update function in the Ethel memo, namely, the use of simple auxiliary functions whose names and usage capture the meanings in the application, and whose definitions can easily be changed. The following is a typical example from that version of the program:

```
(defun action-expression-of (a)(caddr a))
   ;; Extracts the action expression in the action given as argument
   ;; the single argument
   ;; Example: a is (action 5 (let-board lh-12))
   ;; and the value is (let-board lh-12)

(defun verb-of (ax)(car ax))
   ;; Extracts the action verb in an action expression given as
   ;; argument
```

This approach achieves one of the goals mentioned above, namely, it makes it easier to understand the meaning of a limited section of program code when reading it. Compare the following segments from version 1 and version 2 of the program, respectively:

```
(case
   (car (caddr a))
   (let-board
      (let ((x (assoc (cadr (caddr a)) *ap)))
```

against

```
(let ((ax (action-expression-of a)))
     (case (verb-of ax)
       (let-board
          (let* ((x (assoc (arg1-of ax) *ap))
```

The second version is an improvement over the first one, in terms of readability, but it is still a bit primitive. It forces the programmer and the reader to use a number of small, special-purpose functions, and the support for change of representation is somewhat limited. The real importance of being able to write `verb-of` instead of `car` can be questioned. The experienced Lisp user, who is well accustomed to the idea that the first element of a list often contains an operator and the rest of the list contains arguments for that operator, may anyway interpret `car` as "obtain the operator leading the expression" and may not need additional mnemonics to understand that.

Let us then proceed to the main topic of the present lecture note, which is the use of a data-abstraction notation that is implemented using a macro facility.

4 Choice of notation

We shall approach this problem by first designing a notation that is both in the spirit of data abstraction, and in line with the general structure of Lisp. Then in the next section we show how it can be implemented with very little effort.

The following is an example of the notation as applied to the example above. We first declare the structure of the kinds of data structures that are involved in the example, as follows:

```
(declare-struc '(
   (action
       (action (= start)(= actexpr)))
   (action-expression
       ((= verb)(? arg1)(? arg2)(? arg3))
        (let-board (= plane)) )
   (position-expression
       ((= position-type)(= arg1)(? arg2))
        (at-airport (= locn))
        (boarding (= locn)(= tick))
        (ready-to-fly (= locn)) )
       ))
```

The idea is as follows. The declaration specifies the structure of three data types, called `action`, `action-expression`, and `position-expression`, respectively. An object of type `action` is a list of three elements, where the first element must be the symbol `action` and the second and the third elements are variable and are given the names `start` and `actexpr`, respectively. The list beginning with an equality sign indicates an obligatory component.

The declarations for the types `action-expression` and `position-expression` are more complicated, reflecting the fact that these kinds of expressions have subtypes that are characterized by their first element. For example, an object of type `position-expression` must be a list, where the first element can be either `at-airport`, `boarding`, or `ready-to-fly`, and the structure of the rest of the list depends on what the first element is.

Besides the alternatives for each of the possible choices of the first element in the list, there is also one alternative that is used for naming the first element. If one considers an object of type `position-expression` in general, where the choice of first element is not known, then the first element will be called `position-type`, the second element is also obligatory and is called `arg1`, and the third element is conditional (in the sense that it may or may not occur) and is called `arg2` if it occurs. The question-mark indicates that it is conditional.

The declaration for `action-expression` is analagous. For both position-expressions and action-expressions it is of course intended that more cases shall be added, but the cases shown here are sufficient for the re-representation of the state update function given initially.

Given these declarations, we rewrite the state update function for `let-board` as follows:

```
(expd '(defun update-state5 ()
  (setq *t (+ 1 *t))
```

```
(dolist (a *oa)
    ;; example: a is (action 5 (let-board lh-12))
    ;; where lh-12 is airplane; 5 was the starting time
  (with ((a action))
        ;; this binds start and actexpr
    (with ((actexpr action-expression))
          ;; this binds verb, arg1, ...
      (withcase verb actexpr
         ;; this selects the case and binds e.g.
         ;; as (let-board plane) in each case
          (let-board
            (let ((planepos (assoc-val plane *ap)))
                 (with ((planepos position-expression))
                  ;; example of value: (at-airport stockholm)
                  ;; this binds (position-type arg1 arg2 ...)
                    (withcase position-type planepos
                     ;; this withcase binds e.g. (at-airport locn)
                     ;; or (boarding locn tick), etc
                          (at-airport
                            (smash planepos (list 'boarding locn 1)) )
                          (boarding
                             (cond ((equal tick 3)
                                    (remaction actexpr)
                                    (smash planepos
                                        (list 'ready-to-fly
                                              locn )))
                                   (t (setz tick (+ 1 tick))) ))
                          (t nil) ))) )

       ;; Additional action verbs are implemented by adding more
       ;; branches here for the case-expression
          (t nil) ))))

(logstate) ))
```

Notice that the new notation here is not part of CommonLisp; it is shown as an example of what the programmer can easily do himself. Three operators are introduced: `with`, `withcase`, and `setz`. In addition the entire definition is wrapped by the function `typemod`.

The operator `with` is the most significant one, and it works as follows: an expression

```
(with ((object typename)) expression)
```

means that the value of `object` shall be a list that is interpreted as an object of type `typename`, so that the names for elements of the list according to the declarattions are bound to the actually occurring elements. For example, if the value of `ax` is the list `(action 163 (let-board lh-12))` then the value of

```
(with ((ax action)) (list (cadr actexpr) start))
```

will be

```
(lh-12 163)
```

because following the declarations, `start` is bound to `163` and `actexpr` is bound to `(let-board lh-12)`.

Moreover, using the general idea of implicit `progn`, an expression of the form

```
(with ((object typename)) expr1 expr2 ... exprn)
```

is evaluated by evaluating `expr1`, ... `exprn` in succession with the bindings obtained from `((typename object))`, and returning the value of `exprn` as the value of the entire `with`-expression.

The operator `withcase` is a combination of `with` and `case` and works as follows. For the expression

```
(withcase position-type planepos
  (at-airport expr-1)
  (boarding expr-2)
  (t nil) ))))
```

the program will behave according to the schema

```
(case position-type
  (at-airport expr-1)
  (boarding expr-2)
  (t nil) ))))
```

but in the evaluation of one of the branches `expr-i` it binds as variables the field names of the appropriate variant of the structure definition for the type to which the second "argument" of `withcase` belongs. In the example, if `expr-1` is to be evaluated, one uses the following case of the definition for `position-expression`

```
(at-airport (= locn))
```

so that `locn` is bound to the second element of the list that is the value of `planepos`. Similarly, if `expr-2` is to be evaluated one uses

```
(boarding (= locn)(= tick))
```

and binds both `locn` and `tick`.

Looking over the entire definition one can see that the binding of variables using the `with` and `withcase` operators is used recursively. For example, the expression

```
(with ((action a)) ...)
```

binds `actexpr` which occurs in the subsequent `withcase` expression, and so on further down into the entire definition.

5 Implementation

The implementation of the operators mentioned above has been made and used for the example shown here and a few others. A description of the implementation and its restrictions will be added here; for the time being the comments in the program are the only source of detailed information.