Advanced Functional Programming, 1DL450

Lecture 6, 2012-11-16

Cons T Åhs

Caching revisited

- In Erlang we wrote a cache using the state of a process.
- Another obvious way of writing a cache is to store the cache/state in a variable that survives between function calls
- We want a general solution that can be used for any function
- The state should not be shared between different cached functions
- Define a macro `DEFCACHEFUN` that can be used instead of `DEFUN`
- `LET` can be used surrounding a `DEFUN`
  - A permanent context is created that is accessible in the `DEFUN`

```
(let ((outer nil))
  (defun foo (x y z)
      ... outer ..
      (setf (.. outer) ..)))
```

Caching revisited

```
(defmacro defcachefun (name args &rest body)
  (let ((cache (gensym))
        (value (gensym))
        (exists (gensym))
        (actualargs (gensym)))
    `(let ((,cache (make-hash-table :test #'equal)))
       (defun ,name (&rest ,actualargs)
         (multiple-value-bind (,value ,exists)
             (gethash ,actualargs ,cache)
           (if ,exists
               ,value
               (destructuring-bind ,args ,actualargs
                 (setf (gethash ,actualargs ,cache)
                       (progn ,@body)))))))))
```

- Extensive use of `GENSYM` to safe guard free names in body
- Note use of `DESTRUCTURING-BIND`; allows complex argument lists (with `&KEY`, `&REST` etc)
- We are still defining the same function name, so recursive calls will call the cached function and benefit as well

Caching revisited

```
(defcachefun fib (n)
          (cond ((= n 0) 0)
                ((= n 1) 1)
                (t (+ (fib (- n 2)) (fib (- n 1))))))

;; expands to

(LET ((#2=#:G311274 (MAKE-HASH-TABLE :TEST #'EQUAL)))
  (DEFUN FIB (&REST #1=#:G311277)
    (MULTIPLE-VALUE-BIND (#4=#:G311275 #3=#:G311276)
        (GETHASH #1# #2#)
      (IF #3#
          #4#
        (DESTRUCTURING-BIND (N) #1#
          (SETF (GETHASH #1# #2#)
                (PROGN (COND ((= N 0) 0) ((= N 1) 1)
                       (T (+ (FIB (- N 2)) (FIB (- N 1))))))))))))
```

- Why is it beneficial to order the recursive calls as above?
- Fast computation of large fibonacci numbers
  - trade speed for space

Performance

- Is Lisp slow?
  - Interpreted programs are generally slower than compiled
  - Modern implementations of Lisp (from the 70s and onward) include a compiler to native code.
  - In a mid 70s comparison of Lisp and Fortran for a purely numerical problem, Lisp won. Why?
  - More recent comparisons between Lisp and other languages exist
- Paul Graham says:
  - Lisp is two languages: a language for writing fast programs and a language for writing programs fast.
- A language for writing programs fast is easy to construct
  - high level and rich with features (symbolic, math based, no typing etc)
  - you are more productive in a high level language
  - making the program fast requires a good compiler, but does the language have to be low(er) level?

Performance

- Common Lisp is dynamically typed, meaning that type checks are performed at runtime.
- For compiling it is possible to add declarations and hints to the compiler.
- The declarations are standardised, but exactly how much the compiler uses them is up to the implementation. Some examples
  - `OPTIMIZE` - hint to compiler what to optimise on
    - `COMPILATION-SPEED`
    - `DEBUG` - ease of debugging
    - `SPEED` - speed of object code
    - `SAFTEY` - run time error checking
    - `SPACE` - both code size and run time space
  - `TYPE` - specify type of variable
  - `FTYPE` - specify type of function
  - `INLINE` - hint to compiler that a function can/should be inlined
  - `SPECIAL` - declare name to be dynamically scoped

Performance

- Declarations can be placed either locally (inside functions) or globally
  - locally use `DECLARE`
  - globally use `DECLAIM` or `PROCLAIM` (equivalent)
- Different declarations can affect run time properties such as what happens when you feed a function arguments of the wrong type
  - the code will assume the declaration correct and generate code for that
  - with incorrect declarations or or incorrect types at call time, the code can fail in new ways

Performance

- Code to create a two dimensional array of numbers and then sum all the numbers

```
(defun create-array-0 ()
  (make-array '(1000 1000) :initial-element 1))

(defun sum-array-0 (array)
  (let ((sum 0))
    (dotimes (i 1000)
      (dotimes (j 1000)
        (incf sum (aref array i j))))
    sum))
```

Performance

- Same code, but decorated with declarations of types and optimisation

```
(defun create-array-1 ()
  (make-array '(1000 1000)
              :initial-element 1
              :element-type 'fixnum))

(defun sum-array-1 (array)
  (declare (optimize (speed 3)
                     (space 0)
                     (safety 0)))
  (declare (type (simple-array fixnum (1000 1000)) array))
  (let ((sum 0))
    (declare (type fixnum sum))
    (dotimes (i 1000)
      (declare (type fixnum i))
      (dotimes (j 1000)
        (declare (type fixnum j))
        (incf sum (aref array i j))))
    sum))
```

Performance

```
CL-USER 510 > (compare-sum-array 1000)
Timing the evaluation of (DOTIMES (I N) (SUM-ARRAY-0 ARR-0))

User time    =       20.253
System time  =        0.114
Elapsed time =       20.048
Allocation   = 399288 bytes
57 Page faults
Timing the evaluation of (DOTIMES (I N) (SUM-ARRAY-1 ARR-1))

User time    =        2.984
System time  =        0.018
Elapsed time =        2.947
Allocation   = 48944 bytes
0 Page faults
NIL
```

- Compare the speed using `(TIME expression)`
- Almost 7 times faster, 1/8 memory allocation and no page faults
- Try `(DISASSEMBLE function)`

Performance

```
(defun sum-array-2 (array)
  (declare (optimize (speed 3)
                     (space 0)
                     (safety 0)))
  (declare (type (simple-array fixnum (1000 1000)) array))
  (let ((sum 0))
    (declare (type fixnum sum))
    (dotimes (i 1000)
      (declare (type fixnum i))
      (dotimes (j 1000)
        (declare (type fixnum j))
        (incf sum (aref array j i))))
    sum))
```

change memory access pattern

- One small change can hurt performance a lot

Performance

```
Timing the evaluation of (DOTIMES (I N) (SUM-ARRAY-1 ARR-1))

User time    =        2.964
System time  =        0.021
Elapsed time =        2.935
Allocation   = 39236 bytes
0 Page faults
Timing the evaluation of (DOTIMES (I N) (SUM-ARRAY-2 ARR-1))

User time    =        9.221
System time  =        0.056
Elapsed time =        9.207
Allocation   = 111840 bytes
0 Page faults
```

- Now more than 3 times slower
  - Memory access patterns matter
  - Gives an indication of the quality of the compiler as well

CLOS

Alan Kay, inventor of Smalltalk:

"I invented the term *object oriented*, and I can tell you that C++ wasn't what I had in mind."

- There is more than one way to construct an object oriented programming language
  - Java/C++ and more..
    - define classes with instance variables
    - define methods on those classes; methods are connected to a specific class
  - CLOS - Common Lisp Object System
    - define classes with instance variables (called *slots*)
    - define methods not connected to a specific class

CLOS

- `DEFCLASS` is used to define a class
    - simplest form gives a name, inheritance and description of slots
    - allows multiple inheritance (order important)

```
;; Geometric shapes
(defclass shape () ())

;; Things with colour
(defclass coloured ()
  ((colour :accessor colour :initarg :colour)))

(defclass circle (shape)
  ((radius)
   (center)))

(defclass rectangle (shape)
  ((topleft)
   (width)
   (height)))

(defclass coloured-circle (coloured circle) ())

(defclass colour-rectangle (coloured rectangle) ())
```

CLOS

- We can also imagine having different canvases, with different properties for drawing on them

```
;; Things to draw on - general canvas
(defclass canvas () ())

;; Special canvas with vector graphics
(defclass vector-canvas (canvas) ())
```

CLOS

- Instead of tying methods to a class, CLOS introduces the concept of *generic functions*
- `DEFGENERIC` introduces the *generic* version of the function
- similar to an ordinary function, but the definition is spread over several *methods* which contain specialisations on the arguments

```
(defgeneric inside (point object)
  (:documentation "returns true if point is inside object"))

(defgeneric intersects (object1 object2)
  (:documentation "Return true if the objects intersect"))

(defgeneric draw (object canvas)
  (:documentation "Draw OBJECT on CANVAS"))
```

CLOS

- `DEFMETHOD` introduces actual implementations depending on the arguments
- note that specialisation can be done on all arguments
- we also get rid of the problem of determining which class should know about another

```
(defmethod inside (point (object circle))
  ;; code for determining if point is in circle
  )

(defmethod inside (point (object rectangle))
  ;; code for determining if point is in rectangle
  )

(defmethod intersects ((object1 circle) (object2 circle))
  ... )

(defmethod intersects ((object1 circle) (object2 rectangle))
  ... )
```

CLOS

- Methods are ordered according to how specialised they are, so if several methods are applicable, the most specialised is call
- `CALL-NEXT-METHOD` is used, if needed, to call the next (more general or less specialised) method
- `DRAW` is specialised on two arguments

```
(defmethod draw ((object shape) (output canvas))
  (format t "draw object ~w on canvas ~w~%" object output))

(defmethod draw ((object circle) (output canvas))
  (format t "draw circle on canvas~%")
  (call-next-method))

(defmethod draw ((object circle) (output vector-canvas))
  (format t "draw circle on vector-canvas~%")
  (call-next-method))

(defmethod draw ((object coloured) (output canvas))
  (format t "draw with colour: ~s~%" (colour object))
  (call-next-method))
```

CLOS

```
(setf *circle* (make-instance 'circle))
(setf *canvas* (make-instance 'canvas))
(setf *vector-canvas* (make-instance 'vector-canvas))
(setf *coloured-circle* (make-instance 'coulored-circle
                                       :colour "red"))
```

- Create some instance of our classes
- Note use of keyword argument to for colour - specified in class definition
    - more cumbersome without

CLOS

```
CL-USER 560 > (draw *circle* *canvas*)
draw circle on canvas
draw object #<CIRCLE 41009CBB> on canvas #<CANVAS 20094A3F>
NIL

CL-USER 567 > (draw *circle* *vector-canvas*)
draw circle on vector-canvas
draw circle on canvas
draw object #<CIRCLE 20098D5F> on canvas #<VECTOR-CANVAS 200F395B>
NIL

CL-USER 568 > (draw *coloured-circle* *canvas*)
draw with colour: "red"
draw circle on canvas
draw object #<COULORED-CIRCLE 201036B3> on canvas #<CANVAS 20094A3F>
NIL

CL-USER 569 > (draw *coloured-circle* *vector-canvas*)
draw with colour: "red"
draw circle on vector-canvas
draw circle on canvas
draw object #<COULORED-CIRCLE 201036B3> on canvas #<VECTOR-CANVAS
200F395B>
NIL
```

CLOS

- Methods can also be qualified with `:BEFORE`, `:AFTER`, `:AROUND` to create different kinds of wrappers
- A set of `DEFGENERIC`s can be considered as an interface.
  - implement methods for your specific classes to make them useable in a general setting
- CLOS is not simple
  - multiple inheritance
  - generic functions
  - method chaining
- CLOS is powerful
  - multiple inheritance
  - generic functions
  - method chaining

Summary, Common Lisp

- Main features:
  - program data equivalence
  - macros
- Large language with small core
- Easy to extend the language within the language itself
  - DSLs are easy
  - Programs can focus on the problem domain
- Suitable for quick development, but also for high performance
  - adding declarations about actual types can help a lot
- CLOS
  - another take on object oriented programming
  - multiple inheritance
  - generic functions vs message passing (methods bound to classes)