Vrije Universiteit Brussel

POOL: Python 2.3 style Method Resolution Order in the Scheme macro implementation

January 16, 2011

Roeland Matthijssens, Rolno: 88208, roeland.matthijssens@vub.ac.be

1^{st} Year Master of Science in Applied Sciences and Engineering: Computer Science

Abstract In our course we have seen an implementation of an OO language in scheme using macros. We were tasked to extend this base implementation with multiple inheritance. We have chosen to do this the way python has been doing multiple inheritance since version 2.3. This method is called "C3 Method Resolution Order"

The General Idea

Normaly when we subclass a class we keep a pointer to its superclass. When we don't need a reimplementation of some function in the subclass, we can just not define the function, and expect the superclass to handle the call when we receive it. This is why we keep a pointer to the superclass. When we call a function of a class we look through its the method dictionary to find the method. If it is implemented we can just call it. If its isn't implemented we just call the same function of the superclass (keeping the context in mind) and let the superclass handle it from there. Eventually we will either bump into a superclass that does implement the function, in which case it is executed, or we bump into the root class, in which case an error is raised.
Deligating function calls to the superclass makes sence when we have a clear way of knowing who we want to send the supercall to. This is trivial in a single inheritance setting. But multiple inheritance breaks this easy principal, because the interpretor has no intelligent way of choosing between all the possible superclasses.
So, to help the interpretor, we define an algorithm that, given an inheritance tree, clearly defines the order of the superclasses. This will allow the interpretor to choose between who to deligate the message to. We achieve this ordering by flattening the inheritance tree, using the C3 algorithm.

What To Do With This List

When we want to define a class, instead of providing just one superclass we give a list of classes we want to inherit from. Because these classes also have a possibly long list of superclasses, we can construct complex inheritance trees. Our goal is to flatten this entire tree into a single ordered list.
This list defines the order that the class must follow when he wants to deligate a call to one of its superclasses. It just asks the first class in the list to answer to the call. If this class implements the method, the method resolves. If he doesn't implement the method either, the class that initialy received the message then sends the method to the second class in the list. This process repeats, until we find some class that implements the message, or we bump into the root class.
There are two things to note here. First the ordered list contains not only the classes we provide the definition with, but all the classes in the entire inheritance hierarchy. The second thing to note, is that we will always eventualy bump into the root class, this is a result of including the entire inheritance tree into the flattened list. Because at least some class is a direct subclass of the root, the root will also be included in the tree, and therefor also in the flattened list.
This has the added bonus that we can keep the original way that the system threw the 'method not found' error. Since we will eventually as a last resort ask the rootclass to answer to the call, which will

raise the exeption.

The implementation

Instead of doing the normal recursive supercalls when we can't find the method in the superclass, we need to redefine this so that the lookup for the method in the superclass returns some kind of indication of failure, that represents the absence of the method in the superclass' dictionary. When we receive this failure, we know that we have to send the message to the second superclass.

```
(define-macro MYCLASS
 ...
((<<SINGLETONEVAL>>)
 (let*
   ((context (car args))
    (msg (cadr args))
    (args (caddr args))
    (entry (<<METHODS>> 'get msg)))
   (if entry
    (apply entry (cons context args))
    '<<METHODNOTFOUND>>
    )))
...)))))))
```

This also means that we will have to do this in the root class, which is trivial.

```
(define Root
(lambda (msg . args)
 ...
 ((<<SINGLETONEVAL>>)
  (let*
   ((msg (cadr args)))
   (error "method not found " msg)))
 ...))
```

Instead of calculating the flattened inheritance list from scratch every time we define a new class, we make this list of superclass available in each class. This is the same in languages like python, where you can ask the list of supers at runtime. The advantage of this is that we don't have to recursively generate the flattened list for our parents when we create the list for our new class. We can just take the lists of our parents, and use them to create a new list.

```
(define-macro MYCLASS
(lambda (superlist . defs)
 '(letrec
   ((...
    (<<CLASS>>
     (lambda (msg . args)
      (case msg
       ...
       ((<<GETSUPERS>>) <<SUPERS>>)
       ...))))
 ...)))
```

For bootstrapping we need our rootclass to return an empty list.

```
(define Root
(lambda (msg . args)
 (case msg
  ...
```

```
5    ((<<GETSUPERS>>) (mlist))
6     ...)))
```

Now when we try to evaluate a method in a class, instead of just evaluating, and sending the message to the superclass we try to evaluate it localy. If this fails, we start to loop over our superlist, and the first super that evaluates the method returns the respons.

```
1 ...
2((<<EVAL>>)
3 (let*
4     ((context (car args))
5      (msg (cadr args))
6      (args (caddr args))
7      (entry (<<METHODS>> 'get msg)))
8   (if entry
9       (apply entry (cons context args))
10      (evalWithFirstSuper <<SUPERS>>
11                          context msg args)
12      )))
13 ...
```

evalWithFirstSuper Is basicaly a foldl over the superlist, where we return the first successfull execution.

```
1(define (evalWithFirstSuper superList
2         context msg args)
3 (if (empty? superList)
4  (error "method not found")
5  (let ((result ((mcar superList)
6                 '<<SINGLETONEVAL>>
7                 context msg args)))
8   (if (equal? result
9               '<<METHODNOTFOUND>>)
10   (evalWithFirstSuper
11       (mcdr superList)
12       context msg args)
13    result))))
```

This makes sure that the previously established inheritance order is followed when we call a method. Note that the exeption of an empty list is redundant, because this is handled by the root class.
Having this mechanism we can create a class, and pass it some superclasses as arguments.

```
1(define A (MYCLASS (mlist Y X)))
```

The superlist of our newly created class looks like this:

```
1(define-macro MYCLASS
2 ...
3((<<SUPERS>>
4 (flattenSuperList ,superlist))
5 ...
```

where flattenSuperList calls the actual C3 algorithm, named *merge* in out application. This will result in a single flattened list, which will be the inheritance order for the class.

```
1(define (flattenSuperList lst)
2  (define result (mlist))
3  (define (loop lst)
4   (if (not (empty? lst))
5    (begin
6     (set! result
7      (mappend result
```

```
8         (mlist
9          (mcons
10          (mcar lst)
11          ((mcar lst) '<<GETSUPERS>>))
12         )))
13       (loop (mcdr lst)))))
14   (loop lst)
15   (merge (mappend result (mlist lst))))
```

The implementation of the merge function can be found in section *The Implementation Of C3*. This concludes the method lookup for multiple inheritance. However this is not all that needed to be done. Variable lookup is also an issue when dealing with multiple inheritance. But we just established a proper order for inheritance, this means that we can take advantage of this when we are instantiating our class. Instead of just copying the variable dictionary from the superclass, we loop over all our supers, and in turn let them extend an empty variable dictionary. When one of the parents would put a dupplicate entry in this table, we just ignore it because the class that filled in that variable earlier has a higher preference, since it came first in the inheritance list.

```
1 (define-macro MYCLASS
2  ...
3 (<<VARS>> (mergeTables
4  (mmap (λ(super)
5                    (super '<<COPY>>))
6  <<SUPERS>>)))
7  ...)
```

The function *mergeTables* takes a list of tables, and produces one table, filled with all variables of all the tables, taking their preference into account. The function looks like this.

```
1 (define (mergeTables tableList)
2  (let ((result (<<TABLE>>)))
3   (define (loop tableList)
4    (if (not (empty? tableList))
5     (begin (insertAll (mcar tableList))
6            (loop (mcdr tableList)))))
7   (define (insertAll table)
8    (mmap
9     (λ(entry)
10     (let* ((key (mcar entry))
11            (value (mcdr entry))
12            (present (result 'get key)))
13       (if present
14           'skip
15           (result 'put key value))))
16    (table '<<CONTENT>>)))
17  (loop tableList)
18  result))
```

Because we want to declare variables in subclasses of classes that also declare this variable, we also need to redefine the *VAR*-macro

```
1 (define-macro VAR
2  (lambda (name value)
3   `(if (<<VARS>> 'get ',name)
4     (<<VARS>> 'replace ',name ',value)
5     (<<VARS>> 'put ',name ',value))))
```

this means that a redefinition of a variable just resets the variable. Instead of raising an exception. This has the added effect that we can define the same variable twice in one class definition, which might possibly lead to subtle bugs. This problem however falls out of the scope of this exercise.

The Implementation Of C3

The snippets on previous pages have been reformatted for visual purposes. If these are not readable, we kindly direct you to the source code, which is indented by the default scheme standards.

```
(define (merge seqs)
  (define result (mlist))
  (define (loop seqs)
    (let ((nonemptySeqs (filterEmpty seqs)))
      (if (not (empty? nonemptySeqs))
          (let ((cand (findCandidate nonemptySeqs)))
            (if (not cand)
                (error "inconsistent hierarchy")
                (begin (set! result (mcons cand result))
                       (loop (removeCandidate cand nonemptySeqs))))))))
  (define (filterEmpty seqs)
    (define result (mlist))
    (define (loop seqs)
      (if (not (empty? seqs))
          (let ((head (mcar seqs))
                (tail (mcdr seqs)))
            (if (not (empty? head))
                (set! result (mcons head result)))
            (loop tail))))
    (loop seqs)
    (mreverse result))
  (define (findCandidate seqs)
    (define result false)
    (define (loop seqs)
      (if (not (empty? seqs))
          (let ((head (mcar (mcar seqs))))
            (if (elementOfTails head (mcdr seqs))
                (loop (mcdr seqs));not a good head
                (set! result head))))) ;good head
    (define (elementOfTails element seqs)
      (if (empty? seqs)
          false
          (if (mmember element (mcdr (mcar seqs)))
              true
              (elementOfTails element (mcdr seqs)))))
    (loop seqs)
    result)
  (define (removeCandidate cand seqs)
    (define result (mlist))
    (define (loop seqs)
      (if (not (empty? seqs))
          (begin
            (if (eq? (mcar (mcar seqs)) cand)
                (set! result (mcons (mcdr(mcar seqs)) result))
                (set! result (mcons (mcar seqs) result)))
            (loop (mcdr seqs)))))
    (loop seqs)
    (mreverse result))
  (loop seqs)
  (mreverse result))
```

Full MYCLASS Definition

```
(define-macro MYCLASS
  (lambda (superlist . defs)
    `(letrec
         ((<<SUPERS>> (flattenSuperList ,superlist))
          (<<METHODS>> (<<TABLE>>))
          (<<VARS>> (mergeTables (mmap (λ(super)(super '<<COPY>>))<<SUPERS>>)))
          (<<CLASS>>
           (lambda (msg . args)
             (case msg
               ((new)
                (let*
                    ((context (<<VARS>> 'instantiate))
                     (self
                      (lambda (msg . args)
                        (<<CLASS>> '<<EVAL>> context msg args))))
                  (context 'replace '<<SELF>> self)
                  self))
               ((<<EVAL>>)
                (let*
                    ((context (car args))
                     (msg (cadr args))
                     (args (caddr args))
                     (entry (<<METHODS>> 'get msg)))
                  (if entry
                      (apply entry (cons context args))
                      (evalWithFirstSuper <<SUPERS>> context msg args)
                      )))
               ((<<SINGLETONEVAL>>)
                (let*
                    ((context (car args))
                     (msg (cadr args))
                     (args (caddr args))
                     (entry (<<METHODS>> 'get msg)))
                  (if entry
                      (apply entry (cons context args))
                      '<<METHODNOTFOUND>>
                      )))
               ((<<COPY>>)(<<VARS>> 'copy))
               ((<<GETVARS>>) <<VARS>>)
               ((<<GETSUPERS>>) <<SUPERS>>)
               (else
                (error "invalid class message" msg))))))
       ,@defs
       <<CLASS>>)))
```