Specialization Oriented Programming

Jim Newton,

VCAD, Cadence Design Systems

29th July 2007

Overview

- Introduction
- Definition of SOP
- Examples of generic specializer in application programming.
- Developing a generic specializer, meta programming.
- Conclusion

Background

- What is SKILL/SKILL++?
- What is VCLOS - VCAD Common Lisp-like Object System?

Development of VCLOS

- Multiple dispatch
- Meta-object protocol
- Method parameter precedence
- Method qualifiers: `before`, `after`, `around`
- Generic specializers
  - Equivalence specializers
  - Domain/Application specific specializers

Specialization

Specialization in CLOS

```
(defmethod foo ((v1 LIST) (v2 SYMBOL))
  ...)
(defmethod foo ((v1 (EQL nil)) (v2 (EQL t)))
  ...)
```

Domain Specific Specializers

We would like to be able to specify methods as follows:

```
(defmethod foo ((v1 (SPEC1 data1)) (v2 (SPEC2 data2)))
  ...)
(defmethod foo ((v1 (eql nil)) (v2 (spec1 data1)))
  ...)
```

For example:

```
(defmethod foo ((v1 (EQUAL (1 2 3))) (v2 (? oddp)))
  ...)
```

Problem solving with various OOP approaches:

- Class – *classes encapsulate the problem. Objects are actors manipulating data.*
- Generic Function – *method definitions determine:*
  - *what is called?*
  - *in which order?*
- Specialization Oriented – *domain specific specializers allowing methods to elegantly specify applicability.*

Challenges to implementing specializers

- Identify syntax of a specializer name in a defmethod form.
- Determine which methods are applicable
- Determine order of specificity
- Provide acceptable performance (memoization)

Examples

Example Application Development

Develop a program which will walk Scheme source–warning about unused and unbound variable references.

Traversing Lists

Lists are traversed with updated call-stack.

```
(defmethod Walk ((expr list) env call-stack)
  (let ((call-stack (cons expr call-stack)))
    (dolist (sub expr)
      (Walk sub env call-stack))))
```

All non-lists are ignored by default.

```
(defmethod Walk ((expr t) env call-stack)
   nil)
```

Symbols

Symbols are treated as variable references.
Unbound variables are reported.

```
(defmethod Walk ((var symbol) env call-stack)
   (if-let (binding (find-binding env var))
      (push call-stack (used binding))
      (format t "unbound: ~A: ~A~%" var call-stack)))
```

Quoted lists

CONS specializer prunes traversal into quoted lists.

```
(defmethod Walk ((form (CONS (eql QUOTE)))
                 env
                 call-stack)
   nil)
```

Syntax Examples

- `(CONS number)`
  - list whose first element is a number
- `(CONS (eql 42))`
  - list whose first element is 42
- `(CONS (CONS (eql 42)))`
  - list whose first element is a list whose first element is 42

Building variable bindings

- CONS specializer recognizes `LAMBDA` expression
- parse `LAMBDA` form
- parse lambda-list
- traverse body of `LAMBDA` with extended environment
- report unused variables

Lambda expressions

```
(defmethod Walk ((form (CONS (eql LAMBDA)))
                 env
                 call-stack)
   ... )
```

Parse the lambda form and lambda-list

```
(defmethod Walk ((form (CONS (eql LAMBDA)))
                 env
                 call-stack)

>>  (destructuring-bind (_ lam-list &rest body) form
>>     (let ((bindings (derive-bindings lam-list)))
      ...
      )))
```

Traverse body of lambda with extended environment

```
(defmethod Walk ((form (CONS (eql LAMBDA)))
                 env
                 call-stack)
   (destructuring-bind (_ lam-list &rest body) form
      (let ((bindings (derive-bindings lam-list)))

>>      (let ((env (extend-env bindings env))
>>            (call-stack (cons form call-stack)))
>>         (dolist (form body)
>>            (Walk form env call-stack)))
    ...
    ))))
```

Report unused variables

```
(defmethod Walk ((form (CONS (eql LAMBDA)))
                 env
                 call-stack)
   (destructuring-bind (_ lam-list &rest body) form
      (let ((bindings (derive-bindings lam-list)))
        (let ((env (extend-env bindings env))
              (call-stack (cons form call-stack)))
           (dolist (form body)
              (Walk form env call-stack)))

>>      (dolist (bind bindings)
>>        (unless (used bind)
>>           (format t "unused: ~A: ~A~%"
>>                   var call-stack))))))
```

Analogous to CONS specializers

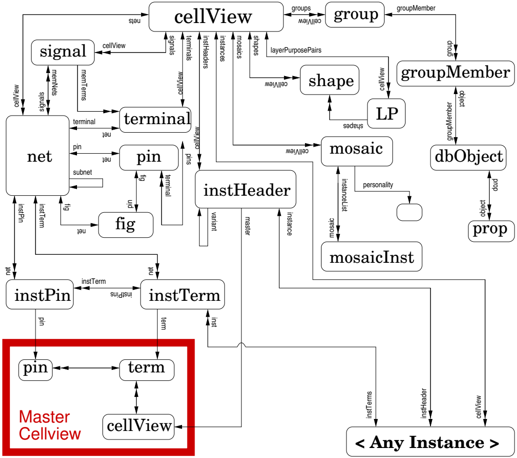
Using the Cadence IC design software, the SKILL programmer

- encounters non-OO objects
- needs to describe their applicablity declaratively

The VCLOS system provides a variety of specializers which enable the programmer to use

- objType specializers
- residual class specializers

CDBA Schema


cellView
groups
cellView
group
groupMember
nets
signal
cellView
signals
terminals
instHeaders
instances
mosaics
shapes
layerPurposePairs
group
groupMember
memNets
memTerms
cellView
shape
cellView
cellView
signals
cellView
terminal
LP
object
shapes
groupMember
terminal
net
net
cellView
mosaic
cellView
pin
net
pins
dbObject
pin
subnet
fig
terminal
instHeader
instanceList
personality
prop
pin
fig
object
instPin
instTerm
variant
master
instance
mosaic
prop
fig
net
mosaicInst
net
net
instTerm
instPin
instTerm
instPins
pin
term
inst
pin
term
Master Cellview
cellView
instTerms
instHeader
cellView
< Any Instance >

objType specializers

- We want to declare (generic) functions that dispatch based on design component: shapes, nets, terminals, etc.
- Cadence database (CDB) is not object oriented, but offers introspective capabilities.
- The *objType* specializer allows method applicability according to the types of object.

Residual class specializers

- *Residual class* specializers are useful for database objects that have been created in the persistent CDB by object oriented programs.
- They determine applicability not on the object's class, but rather on the *policy* class that was used to create the object.
- This is useful because CDB cannot maintain a link to the policy object–which might be out of scope
  - It could have been garbage collected
  - or live in a completely different UNIX process.

Meta Programming

Defining the SOP generic function

To define a new type of specializer, the programmer must use the VCLOS MOP to define several things:

- How to recognize the syntax of a specializer in a method declaration.
- How to compare (sort in order) this type of specializer to other specializers.
- How to compare two specializers of the same type.
- How to determine whether an object matches the specializer.

Define the specializer class

```
(defclass SopConsSpecializer (ClosSpecializer)
  ((enclosedSpecializer
    @initarg enclosedSpecializer
    @reader SopGetEnclosedSpecializer
    @writer SopSetEnclosedSpecializer)
   ...))
```

Define the generic function meta-class

```
(defclass SopConsGenericFunction
          (ClosSpecGenericFunction)
  ())
```

Establish the order of specificity

1. ClosEqvSpecializer (most specific)
2. SopConsSpecializer
3. ClosClassSpecializer (least specific)

```
(defmethod ClosAvailableSpecializers
                    ((gf SopConsGenericFunction))
  ’(ClosEqvSpecializer
    SopConsSpecializer
    ClosClassSpecializer))
```

Identify CONS syntax in ClosDefMethod

```
(ClosDefMethod foo ((v (cons number)))
   ...)
(ClosDefMethod foo ((v (cons (eqv 42))))
   ...)

(foo (list 42))
```

Identify CONS syntax in ClosDefMethod

Return TRUE if `specializer_name` is something like `(cons number)`

```
(defmethod ClosMatchesSpecializerSyntaxP
                   ((specializer SopConsSpecializer)
                    specializer_name)
  (and (listp specializer_name)
       (eq ’cons (car specializer_name))
       (cdr specializer_name)
       (null (cddr specializer_name))
       (ClosNameToSpecializer
               (ClosGetGenericFunction specializer)
               (cadr specializer_name))))
```

Determining applicablity of CONS specializer

```
(ClosDefMethod foo ((v (cons number)))
   ...)
(ClosDefMethod foo ((v (cons (eqv 42))))
   ...)

(foo (list 42))
```

Determining applicablity of CONS specializer

```
(defmethod ClosArgMatchesSpecializerP
              ((spec SopConsSpecializer) arg)
  (and (dtpr arg)
       (ClosArgMatchesSpecializerP
          (SopGetEnclosedSpecializer spec)
          (car arg))))
```

Comparing two CONS specializers

```
(ClosDefMethod foo ((v (cons number)))
   ...)
(ClosDefMethod foo ((v (cons (eqv 42))))
   ...)

(foo (list 42))
```

Comparing two CONS specializers

```
(defmethod ClosCmpLikeSpecializers
                       ((spec1 SopConsSpecializer)
                        spec2
                        gf
                        param
                        spec)
  ...
  (ClosCmpSpecializers gf
                       (SopGetEnclosedSpecializer spec1)
                       (SopGetEnclosedSpecializer spec2)
                       param
                       spec))
```

Specializer Comparitors

Skipping lots of details, *comparitors* are needed to aid in memoization.

Example Comparator

Application:

```
(foo (list 1))
```

Most specific:

```
(ClosDefMethod foo ((bar (eqv (1)))) ...)
```

Applicable?

```
(ClosDefMethod foo ((bar (cons number))) ...)
```

ClosDefComparator

```
(ClosDefComparator ((meth_spec SopConsSpecializer)
                    (arg_spec ClosEqvSpecializer))
  (and (dtpr (ClosGetData arg_spec))
       (ClosArgMatchesSpecializerP
          (SopGetEnclosedSpecializer meth_spec)
          (car (ClosGetData arg_spec)))))
```

Goals of VCLOS

SKILL should include an object system which:

- provides features of CLOS,
- interfaces to existing SKILL++ programs
- enables OO techniques on *pre-existing* non-OO systems
- is extensible for IC application programming

Dual Approaches

Complicated problems are simplified by making appropriate abstractions.

- Mountain to Mohammad approach
    - Make domain data conform to the computer language model.
- Mohammad to the mountain approach
    - Enable the language to express truths about the data.

Summary

SOP in the form of extensible specializers allows programmers to use object oriented techniques on data that does not fit traditional object oriented views.

Questions

Questions? Suggestions? Complaints?