A View of the Past and Future of Objects

William Cook
UT Austin

Dahl-Nygaard Prize lecture
ECOOP 2014
• History
• Enterprise Applications
• Models & Synthesis
• Ensō
OO came from simulation
creating & running models
grocery store
airplane
business
user interface
OO class
grammar
behavior
+
state
A modern definition:

An *object* is a first-class, dynamically dispatched behavior.

Inheritance, state, identity, classes are useful but not essential.
Modification and Self-reference

Client

Problem!
Not all clients are modified
Inheritance:
“Consistently modify a recursive definition”
Killer Application of OO:

GUI frameworks
write classes
instantiate them
to create a GUI model
run the model
OO has won!
many PL researchers hate objects
I guess many of them have never programmed a large extensible system.

Or if they have, it is a type system or a compiler, ... or financial application.
All languages are domain specific
Is OO best for everything?
Example

Enterprise Applications
allow read if grade.student = user

class Schema
  types: Type*
class Type
    name: string
class Primitive < Type
class Class < Type
  fields: Field*
super: Type?

on Create Account
  wait(1 month) >> remind
  | cancel
Modeling issues
• composites
• multiple languages
• multiple semantics
• optimization
• UML
large models have many parts
Model = data instance

not well supported in OO
(or PL)
Say It Once

GUI

Data

Security

Data constraints used in GUI

optimize across tiers
UML models
Object-Oriented Design
Wrong thing!

Should model “solution to problem”, not “OO design of program that solves problem”
Models & Synthesis
Spectrum of programming

How (implementation)  What (Specification)
How (implementation)  

What (Specification)  

Verification
Synthesis

How (implementation)

What (Specification)

Verification
Synthesis

How (implementation) - Restricted Specifications - What (Specification)

Verification Lite - Verification (guided)

Type checking, bug finding, race detection, etc.
Model-Driven Development / Domain-Specific Languages: BNF, SQL, Excel, Datalog, XACML, ...

Verification Lite

Verification (guided)

Type checking, bug finding, race detection, etc.

Verification Lite

Synthesis (guided)

Synthesis Lite

Synthesis

Domain-Specific Specifications

What (Specification)

How (implementation)
Ensō
Requirements (what)

Strategies (how)

Application (Code)

Behavior
Requirements (what)

Small change to Strategies

Very different Code!!!

Behavior
Requirements (what)

Very different Code

Small change to Strategies

Behavior
Requirements (what)

Reify!

Strategies (how)

Application (Code)

Behavior
Requirements

Technical Requirements

Problem Specific

General Strategies

Behavior
Using Managed Data (Ruby)

• Description of data to be managed
  
  \[
  \text{Point} = \{ \ x: \text{Integer}, \ \ y: \text{Integer} \ \} 
  \]

• Dynamic creation based on metadata
  
  \[
  p = \text{BasicRecord.new Point} 
  \]
  
  \[
  p.x = 3 
  \]
  
  \[
  p.y = -10 
  \]
  
  \[
  \text{print } p.x + p.y 
  \]
  
  \[
  p.z = 3 \quad \# \text{error!} 
  \]

• *Factory* BasicRecord: Description\(<\text{T}\> \rightarrow \text{T}\)
Other Data Managers

- Mutability: control whether changes allowed
- Observable: posts notifications
- Constrained: checks multi-field invariants
- Derived: computed fields (reactive)
- Secure: checks authorization rules
- Graph: inverse fields (bidirectional)
- Persistence: store to database/external format

- General strategy for all accesses/updates
- Combine them for modular strategies
Grammars

- Mapping between *text* and *object graph*
- A *point* is written as \((x, y)\)

<table>
<thead>
<tr>
<th>Individual</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>((3, 4))</td>
<td>(P ::= \text{[Point]} \ (&quot;x:int&quot;,&quot;y:int&quot;))</td>
</tr>
</tbody>
</table>

- Notes:
  - Direct reading, no abstract syntax tree (AST)
  - Bidirectional: can parse and pretty-print
  - GLL parsing, *interpreted*!
Door StateMachine

start Opened

state Opened
  on close go Closed

state Closed
  on open go Opened
  on lock go Locked

state Locked
  on unlock go Closed

StateMachine Grammar

```
M ::= [Machine] "start" \start:\states[it] > states:S*
S ::= [State] "state" name:sym out:T*
T ::= [Trans] "on" event:sym "go" to:\states[it]>
```

StateMachine Schema

```
class Machine
  start : State
  states! State*

class State
  machine: Machine
  name # str
  out ! Trans*
  in : Trans*

class Trans
  event : str
  from : State / out
to : State / in
```

A StateMachine Interpreter

```
def run_state_machine(m)
    current = m.start
    while gets
        puts "#{current.name}"
        input = $_.strip
        current.out.each do |trans|
            if trans.event == input
                current = trans.to
                break
            end
        end
end
```
Expression Example

Sample Expression

\[ 3*(5+6) \]

Expression Grammar

\[
\begin{align*}
E & ::= \text{[Add]} \; \text{left}:E \; "+" \; \text{right}:M \; | \; M \\
M & ::= \text{[Mul]} \; \text{left}:M \; "*" \; \text{right}:P \; | \; P \\
P & ::= \text{[Num]} \; \text{val}:\text{int} \; | \; "(" \; E \; "\)"
\end{align*}
\]

Expression Schema

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num</td>
<td>val: int</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>left: Exp</td>
<td>right: Exp</td>
</tr>
<tr>
<td>Mul</td>
<td>left: Exp</td>
<td>right: Exp</td>
</tr>
</tbody>
</table>

An Expression Interpreter

module Eval
operation : eval

def eval_Num(val)
  val
end

def eval_Add(left, right)
  left.eval + right.eval
end

def eval_Mul(left, right)
  left.eval * right.eval
end
end
Everything is a language

Schema Language + Interpreter

Grammar Language + Interpreter

StateMachine Schema

Grammar Schema

StateMachine Grammar

StateMachine DSL
Quad-model

Nontrivial bootstrapping
class Schema
  types: Type*
class Type
    name: string
class Primitive < Type
class Class < Type
  fields: Field*
  super: Type?

class Field
  name: string
  type: Type
  many: bool
  optional: bool

primitive string
primitive bool

(Self-Description)
Schema Grammar

start $S$

$S ::= [Schema] types: T^*$

$T ::= P | C$

$P ::= [Primitive] "primitive" name: sym$

$C ::= [Class] "class" name: sym ("<" S+)? fields: F^*$

$S ::= <root.classes[it]>$


$M ::= "\*" \{ many and optional \}

| "?" \{ optional \}

| "+" \{ many \}

$A ::= "/" inverse: <this.type.fields[it]>

| "/" computed: Expr
Grammar Grammar

start G
G ::= [Grammar] "start" start:/rules[it]> rules:R*
R ::= [Rule] name:sym "::=" arg:A
A ::= [Alt] alts:C+ @"|"
C ::= [Create] "[" name:sym "]" arg:S | S
S ::= [Sequence] elements:F*
F ::= [Field] name:sym ":" arg:P | P
P ::= [Lit] value:str
  | [Value] kind:("int" | "str" | "real" | "sym")
  | [Ref] "<" path:Path ">"
  | [Call] rule:/rules[it]>
  | [Code] "{" code:Expr "}"
  | [Regular] arg:P "*" Sep? { optional && many }
  | [Regular] arg:P "?" { optional }
  | "(" A ")"
Sep ::= "@" sep:P
class Grammar    start: Rule    rules: Rule*
class Rule       name: str     arg: Pattern
class Pattern

class Terminal < Pattern
class Lit < Terminal     value: str
class Value < Terminal    kind: str
class Ref < Terminal      path: Expr
class Alt < Pattern       alts: Pattern+
class Sequence < Pattern elements: Pattern*
class Call < Pattern      rule: Rule
class Create < Pattern    name: str arg: Pattern
class Field < Pattern     name: str arg: Pattern
class Regular < Pattern   arg: Pattern sep: Pattern ?
    optional: bool many: bool
Diagrams

• Model
  • Shapes and connectors
• Interpreter
  • Diagram render/edit application
  • Basic constraint solver
Stencils

• **Model: mapping object graph → diagram**
• **Interpreter**
  • Inherits functionality of Diagram editor
  • Maps object graph to diagram
    – Update projection if objects change
  • Maps diagram *changes* back to object graph
• **Binding for data and collections**
  – Strategy uses schema information
  – Relationships get drop-downs, etc
  – Collections get add/remove menus
Schema Diagram Editor
```javascript
diagram(schema)
graph [font.size=12, fill.color=(255,255,255)] {
    for "Class" class : schema.classes
        label class
        box [line.width=3, fill.color=(255,228,181)] {
            vertical {
                text [font.size=16, font.weight=700] class.name
                for "Field" field : class.defined_fields
                    if (field.type is Primitive)
                        horizontal {
                            text field.name // editable field name
                            text "": 
                            text field.type.name // drop-down for type
                        }
            }
        }
    }
}
```
// create the subclass links
for class : schema.classes
  for "Parent" super : class.supers
    connector [line.width=3, line.color=(255,0,0)]
      (class --> super)

[also for relationships]
Language Workbench Challenge

• Models
  • Physical heating system
    – furnace, radiator, thermostat, etc
  • Controller for heating system

• Interpreter
  • Simulator for heating system
    – pressure, temperature
  • State machine interpreter
    – Events and actions
Physical Heating System Model
Piping Controller

START
START\_GAS: 50
Pump\_flow: 0.35
WATER\_MARGIN: 3
RADIATOR\_MARGIN: 3
BURNER\_RAMPUP: 2
BURNER\_RAMPDOWN: 2

IGNITE
Set Burner\_ignite to true
Set Burner\_gas\_level to START\_GAS
Set Pump\_run to true
Turn splitter Valve center

RAMPUP
Raise Burner\_gas\_level by BURNER\_RAMPUP
Turn splitter Valve center

RUNNING

BOILER
Raise Burner\_gas\_level by BURNER\_RAMPUP
Turn splitter Valve left

COOLDOWN
Lower Burner\_gas\_level by BURNER\_RAMPDOWN

RADIATOR
Raise Burner\_gas\_level by BURNER\_RAMPUP
Turn splitter Valve right
Performance

• Ensō is currently slow but usable
  • Accessing a field involves two levels of meta-interpretation
  • My job is to give compiler people something to do

• Partial Evaluation of model interpreters

  \[ \text{web}((\text{UI}, \text{Schema}, \text{db}, \text{request}) : \text{HTML} \]

  \[ \text{web}[[\text{UI}, \text{Schema}](\text{db}, \text{request}) : \text{HTML} \]

  static dynamic
<table>
<thead>
<tr>
<th>Aspect</th>
<th>Code SLOC</th>
<th>Model SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootstrap</td>
<td>387</td>
<td>—</td>
</tr>
<tr>
<td>Utilities</td>
<td>256</td>
<td>—</td>
</tr>
<tr>
<td>Schemas</td>
<td>691</td>
<td>51</td>
</tr>
<tr>
<td>Grammar/Parse</td>
<td>885</td>
<td>106</td>
</tr>
<tr>
<td>Render</td>
<td>318</td>
<td>17</td>
</tr>
<tr>
<td>Web</td>
<td>932</td>
<td>305</td>
</tr>
<tr>
<td>Security</td>
<td>276</td>
<td>46</td>
</tr>
<tr>
<td>Diagram/Stencil</td>
<td>1389</td>
<td>176</td>
</tr>
<tr>
<td>Expressions</td>
<td>448</td>
<td>144</td>
</tr>
<tr>
<td><strong>Core</strong></td>
<td><strong>5582</strong></td>
<td><strong>844</strong></td>
</tr>
<tr>
<td>Piping</td>
<td>527</td>
<td>268</td>
</tr>
</tbody>
</table>
Ensō Summary

• Executable Specification Languages
  • Data, grammar, GUI, Web, Security, Queries, etc.
• External DSLs (not embedded)
• Interpreters (not compilers/model transform)
  • Multiple interpreters for each languages
• Composition of Languages/Interpreters
  • Reuse, extension, derivation (inheritance)
• Self-implemented (Ruby for base/interpreters)
  • Partial evaluation for speed
Don't Design Your Programs
Program
Your Designs

Ensō
enso-lang.org