Ensō

William Cook, Alex Loh
UT Austin
Tijs van der Storm
CWI
Prevent Bad

Enable Good
Bug Finding
Race Detection
Type Checking
etc.

Prevent Bad

Enable Good
Bug Finding
Race Detection
Type Checking
etc.

Prevent Bad

Enable Good
New languages?
New features?
For what?
Bug Finding
Race Detection
Type Checking
etc.

Prevent Bad

Advantages:
Measurable
Domain-free

Enable Good

New languages?
New features?
For what?
Kolmogorov Complexity
Shortest program that generates information
Best program that generates information behavior
Qualitative Kolmogorov Program Complexity
I don't know how
but it’s a good goal
A Problem

1. Many (many!) repeated instances of similar code

2. Unique details and names prevent generalization
Requirements (what)

Strategies (how)

Application (Code)

Behavior
Small change to Requirements

Very different Strategies

Very different Code

Behavior
Small change to Requirements

Very different Strategies

Chaos!

Very different Code

Behavior
Requirements (what)

Strategies (how)

Application (Code)

Behavior

Reify!?
Data Requirements

Technical Requirements

Data Model

Data Manager

Objects
Using Managed Data (Ruby)

• Description of data to be managed
  
  `Point = { x: Integer, y: Integer }`

• Dynamic creation based on metadata
  
  `p = BasicRecord.new Point`
  `p.x = 3`
  `p.y = -10`
  `print p.x + p.y`
  `p.z = 3  # error!`

• *Factory* `BasicRecord`: `Description<T> ➔ T`
Implementing Managed Data

• Override the "dot operator" (p.x)

• Reflective handling of unknown methods
  • Ruby method_missing
  • Smalltalk: doesNotUnderstand
  • Also IDispatch, Python, Objective-C, Lua, CLOS
  • Martin Fowler calls it "Dynamic Reception"

• Programmatic method creation
  • E.g. Ruby define_method

• Partial evaluation
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
Graphs, Invariants, Computed

Constraints: for all student s

\[ s.\text{dept} = s.\text{advisor.dept} \]

Computed values/attribute grammars
Traditional Data Mechanisms
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 ::= [Point] &quot;(&quot; x:int &quot;,&quot; y:int &quot;)&quot;</td>
</tr>
</tbody>
</table>

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

**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**

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

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

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

**A StateMachine Interpreter**

```ruby
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
end
```
Expression Example

Sample Expression

3*(5+6)

Expression Grammar

E ::= [Add] left:E "+" right:M | M
M ::= [Mul] left:M "*" right:P | P
P ::= [Num] val:int | "(" E ")"

Expression Schema

class Exp

class Num
  val : int

class Add
  left : Exp
  right : Exp

class Mul
  left : Exp
  right : Exp

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
Grammar Grammar

start G
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
Everything is a language

Schema Language + Interpreter

Grammar Language + Interpreter

StateMachine DSL
Quad-model

Instance of

Schema
Schema

Grammar
Schema

Instance of

Schema
Grammar

Instance of

Grammar
Grammar

Instance of

Formatted by

Nontrivial
bootstrapping

Formatted by

Formatted by

Formatted by
class Schema
    types: Type*

class Type
    name: string

class Primitive < Type
    primitive string
    primitive bool

class Class < Type
    fields: Field*
    super: Type?

(Self-Description)
Diagrams

• Model
  • Shapes and connectors

• Interpreter
  • Diagram render/edit application
  • Basic constraint solver
Schema Diagram
Stencils

• Model: mapping object graph $\rightarrow$ 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
Schema Stencil

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

BOILER
Raise Burner.gas_level by BURNER_RAMPUP
Turn splitter Valve left

RUNNING

RADIATOR
Raise Burner.gas_level by BURNER_RAMPUP
Turn splitter Valve right

COOLDOWN
Lower Burner.gas_level by BURNER_RAMPDOWN
Piping Details

- Simulation updates physical model
  - Change to physical model causes update to view
  - Observable Data Manager -> Presentation update
- State machine interpreter changes states
  - Presentation shows current state
- User can interact with physical model
  - Change thermostat
- User can edit diagram
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
  \[
  \text{web}(\text{UI, Schema, db, request}) : \text{HTML} \\
  \text{web}[\text{UI, Schema}](\text{db, request}) : \text{HTML}
  \]
<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>Core</td>
<td>5582</td>
<td>844</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 embeded)
• 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
Related Work

• Aspects: a fundamental idea
  • Current solutions are terrible (AspectJ)
• DSLs and Models: Feeling same elephant
  • external vs. internal
  • graphical vs. textual
• F# Type Providers
• Scheme macros (defstruct)
• Metaprogramming
  • But without manipulating 'code'
Spectrum of programming

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

Verification
How (implementation) => Domain-Specific Specifications => What (Specification)

Verification Lite => Verification

Verification Lite = Type checking
Synthesis Lite = Model-Driven Development Domain-Specific Languages, ...

How (implementation)  \[ \text{Synthesis Lite} \]  \[ \text{Synthesis (guided)} \]  \[ \text{Domain-Specific Specifications} \]  \[ \text{What (Specification)} \]

Verification Lite

Verification
Don't Design
Your Programs
Program
Your Designs

Ensō
enso-lang.org