Ensō

William Cook & *Tijs van der Storm*

UT at Austin

William Cook

Ensō

Ensō (円相) is a Japanese word meaning "circle" [...]

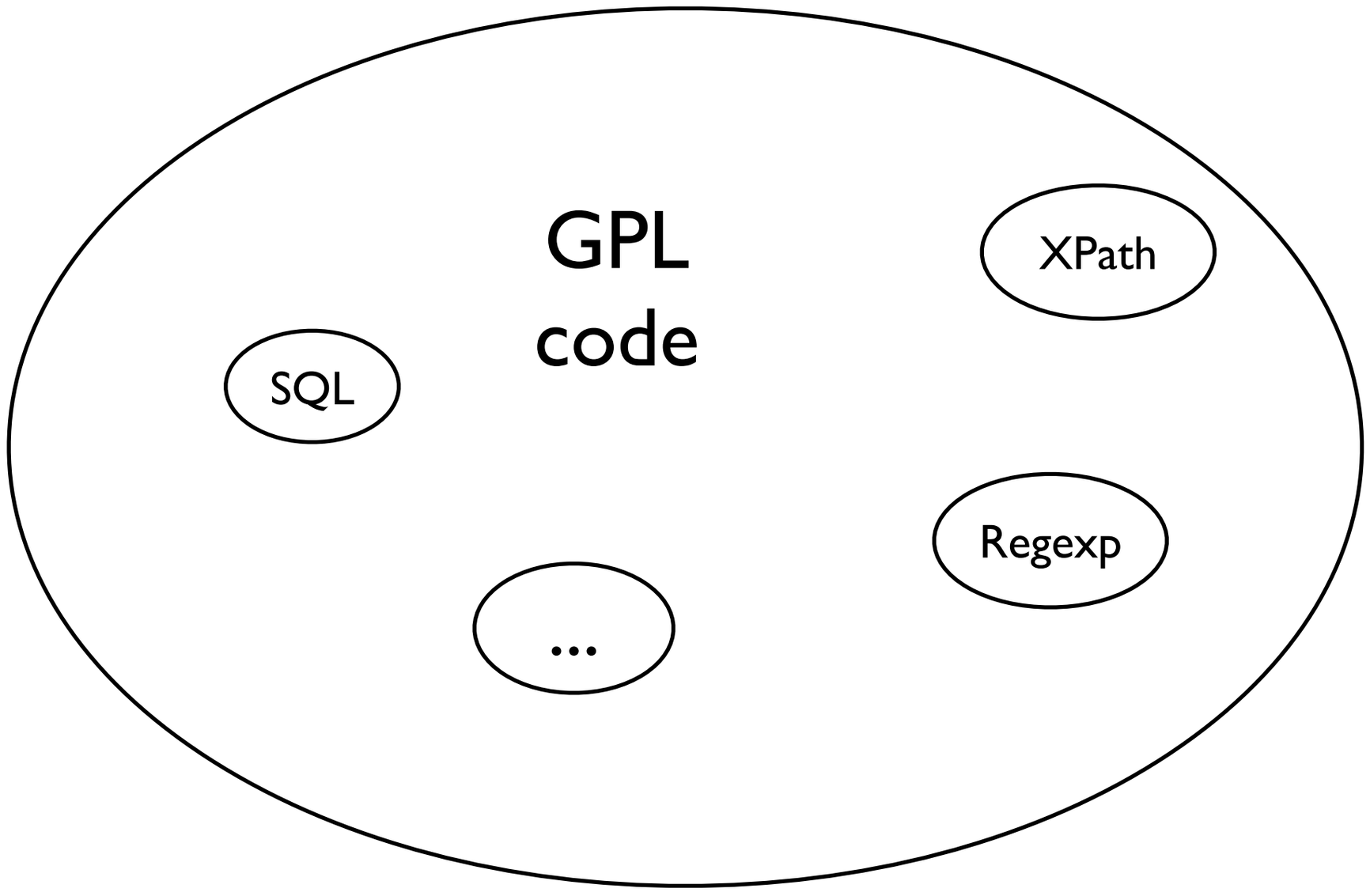
It symbolizes the Absolute enlightenment, strength, elegance, the Universe, and the void [...].

(Wikipedia)

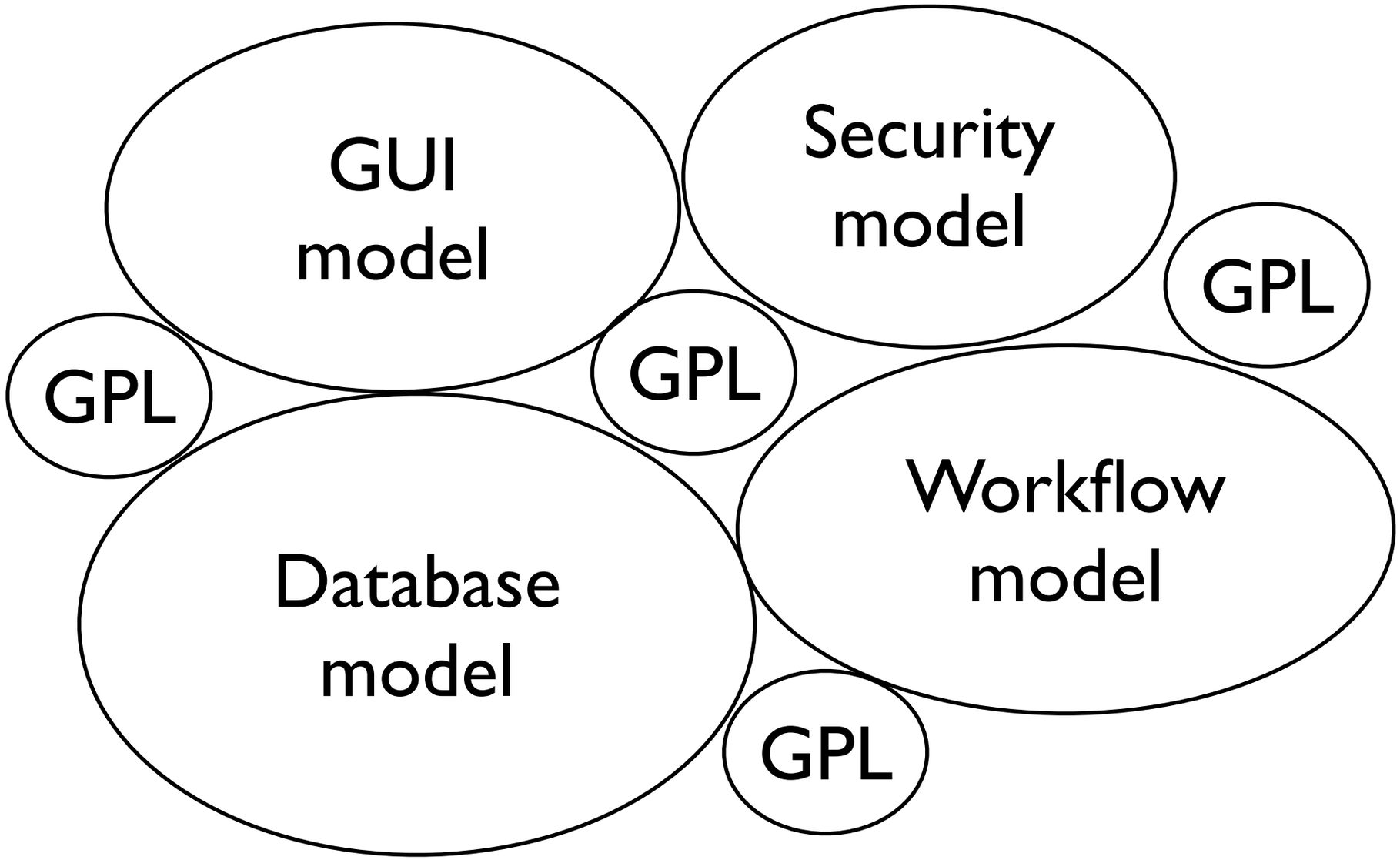
What is Ensō?

- A language workbench?
- A library?
- A design pattern?
- A style?
- ...?

Much GPL little DSL



Much DSL little GPL



etc...

What vs how

- What:
  - Data, Security, GUI, Workflow etc.
- How:
  - “Strategies” / “Designs”
- Tangling/scattering
- Don’t design your programs, program your designs!

Ensō Goals

- Build apps
  - Business, IDE, Spreadsheet, Email etc.
- Scrap more boilerplate
- Solution for cross-cutting concerns (?)
- “Smalltalk of modeling”

How?

- Models + Interpreters = Software
- Problem domains as *information models*
  - (Ensō model = object-graph/ “database”)
- Interpret models using *code*
  - (currently we use *Ruby*-- translated to JS)
- *Model-driven programming system*

(Anti-)Slogans

- No code generation
- No “everything is a ...”
- (Cyclic) graphs are useful
- Implemented in itself (meta-circular)
- Don’t hate code
- Objects below the surface
- Partial evaluation
- Text vs diagrams = moot

Schemas

Schemas

Optional field

Schemas

Many field

Optional field

Schemas

Spine field

Many field

Optional field

Schemas

Spine field

Key field

Many field

Optional field

Schemas

Spine field

Key field

Inverse relation

Optional field

Many field

Creating Models

load the schema of schemas

define a "Points" schema

create a Point using the Point factory

```
> ss = Loader.load('schema.schema')
=> <Schema 1267>

> f = Factory.new(ss)
> ps = Loader.load_text('schema', f, "
> primitive int
> class Point
>    x: int
>    y: int
> end")
=> <Schema 4283>

> pf = Factory.new(ps)
> p = pf.Point(1, 2)
=> <Point 4768>

> Print.print(p)
Point
  x: 1
  y: 2
```

Grammars

```
import impl.grammar

start Schema

Schema ::= [Schema] types:TypeDef* @/2

TypeDef ::= Primitive | Class

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

Class ::= [Class] "class" name:sym ClassAnnot /> defined_fields:Field* @/ </
ClassAnnot ::= Parent?
Parent ::= "<" supers:Super+ @","
Super ::= <root.classes[it]>

Field ::= [Field] name:sym.Kind type:<root.types[it]> Multiplicity? Annot?

Kind ::= "#" { key == true }
       |  "##" { (key == true) and (auto == true) }
       |  "!" { traversal == true }
       |  ":"

Multiplicity ::= ."*" { (many == true) and (optional == true) }
           | ."?" { optional == true }
           | ."+" { many == true }

Annot ::= "/" inverse:<this.type.fields[it]>
        | "=" computed:Expr
```

Constructors

Grammars

```
start Schema

Schema ::= [Schema] types:TypeDef* @/2

TypeDef ::= Primitive | Class

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

Class ::= [Class] "class" name:sym ClassAnnot /> defined_fields:Field* @/ </
ClassAnnot ::= Parent?
Parent ::= "<" supers:Super+ @","
Super ::= <root.classes[it]>

Field ::= [Field] name:sym.Kind type:<root.types[it]> Multiplicity? Annot?

Kind ::= "#" { key == true }
       |  "##" { (key == true) and (auto == true) }
       |  "!" { traversal == true }
       |  ":"

Multiplicity ::= ."*" { (many == true) and (optional == true) }
           | ."?" { optional == true }
           | ."+" { many == true }

Annot ::= "/" inverse:<this.type.fields[it]>
        | "=" computed:Expr
```

Grammars

Constructors

Field assignments

```
start Schema

Schema ::= [Schema] types:TypeDef* @/2

TypeDef ::= Primitive | Class

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

Class ::= [Class] "class" name:sym ClassAnnot /> defined_fields:Field* @/ </
ClassAnnot ::= Parent?
Parent ::= "<" supers:Super+ @","
Super ::= <root.classes[it]>

Field ::= [Field] name:sym.Kind type:<root.types[it]> Multiplicity? Annot?

Kind ::= "#" { key == true }
       |  "##" { (key == true) and (auto == true) }
       |  "!" { traversal == true }
       |  ":"

Multiplicity ::= ."*" { (many == true) and (optional == true) }
           | ."?" { optional == true }
           | ."+" { many == true }

Annot ::= "/" inverse:<this.type.fields[it]>
        | "=" computed:Expr
```

Grammars

Constructors

Field assignments

References

```
start Schema

Schema ::= [Schema] types:TypeDef* @/2

TypeDef ::= Primitive | Class

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

Class ::= [Class] "class" name:sym ClassAnnot /> defined_fields:Field* @/ </
ClassAnnot ::= Parent?
Parent ::= "<" supers:Super+ @","
Super ::= <root.classes[it]>

Field ::= [Field] name:sym.Kind type:<root.types[it]> Multiplicity? Annot?

Kind ::= "#" { key == true }
       |  "##" { (key == true) and (auto == true) }
       |  "!" { traversal == true }
       |  ":"

Multiplicity ::= ."*" { (many == true) and (optional == true) }
           | ."?" { optional == true }
           | ."+" { many == true }

Annot ::= "/" inverse:<this.type.fields[it]>
        | "=" computed:Expr
```

Grammars

Constructors

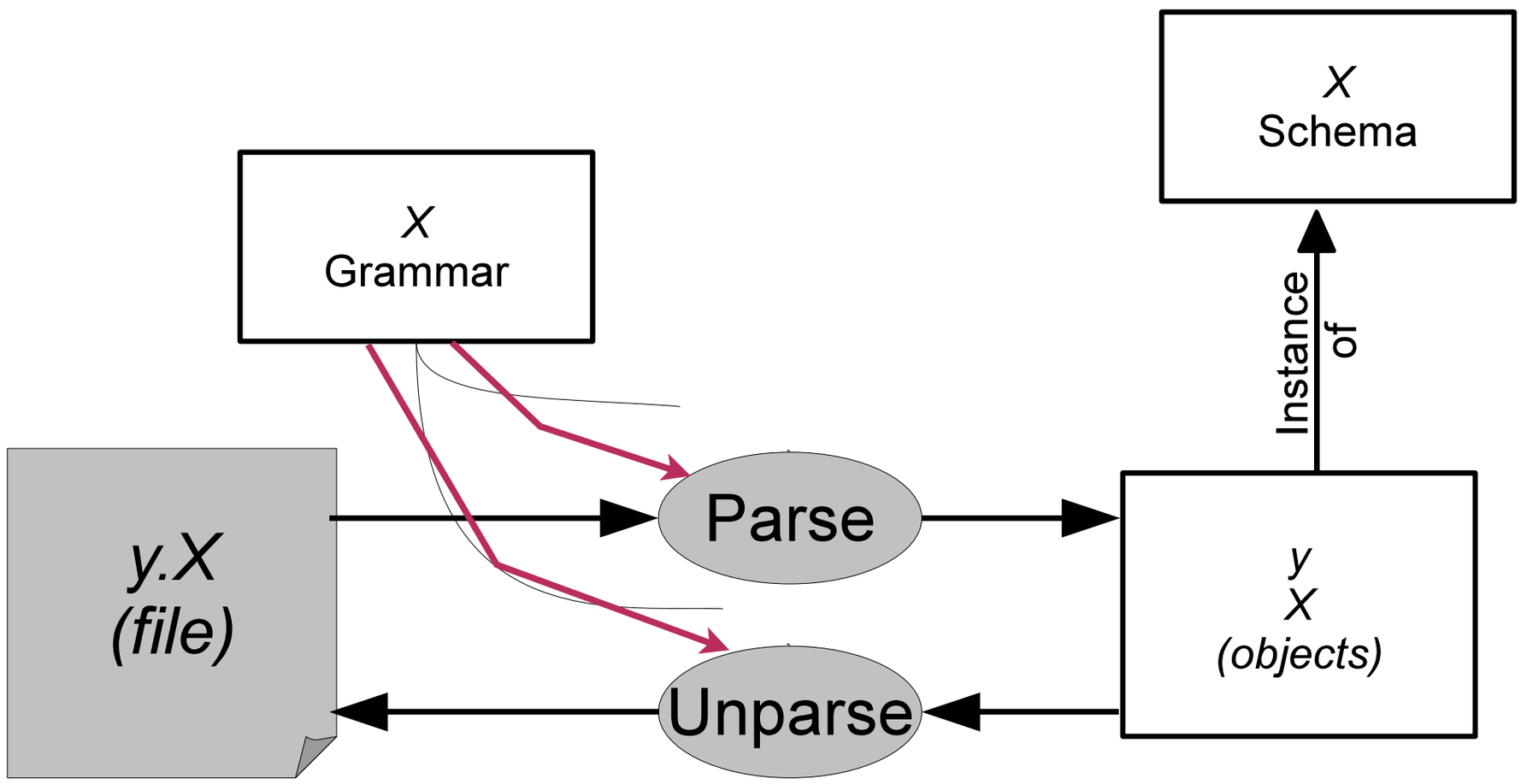
Field assignments

References

"Code"

```
start Schema

Schema ::= [Schema] types:TypeDef* @/2

TypeDef ::= Primitive | Class

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

Class ::= [Class] "class" name:sym ClassAnnot /> defined_fields:Field* @/ </
ClassAnnot ::= Parent?
Parent ::= "<" supers:Super+ @","
Super ::= <root.classes[it]>

Field ::= [Field] name:sym.Kind type:<root.types[it]> Multiplicity? Annot?

Kind ::= "#" { key == true }
       |  "##" { (key == true) and (auto == true) }
       |  "!" { traversal == true }
       |  ":"

Multiplicity ::= ."*" { (many == true) and (optional == true) }
           | ."?" { optional == true }
           | ."+" { many == true }

Annot ::= "/" inverse:<this.type.fields[it]>
        | "=" computed:Expr
```

Parse into instance (not tree/forest)



Rendering

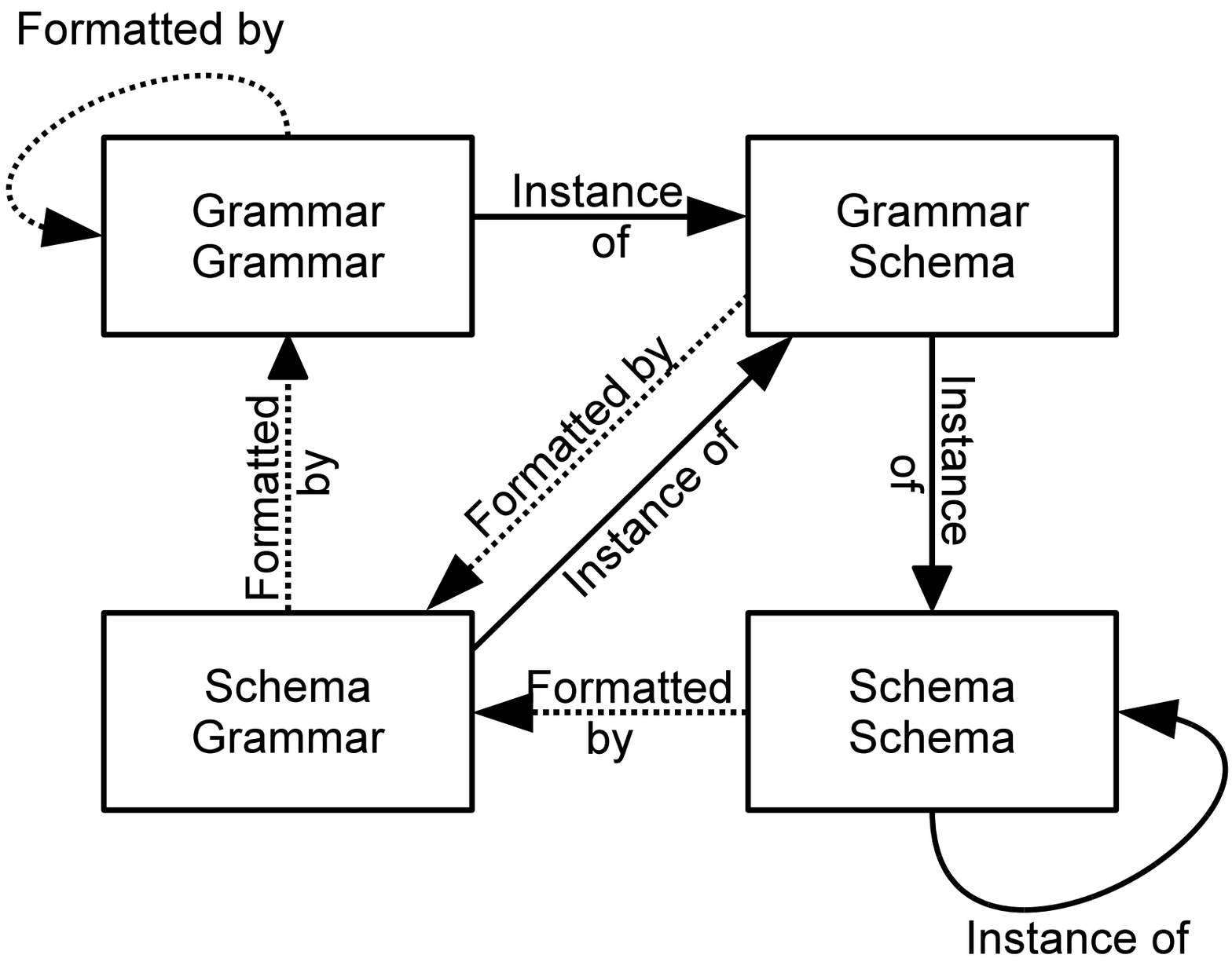
load the schema of grammars

make a grammar of Points

render Point *p* using the grammar

```
> gs = Loader.load('grammar.schema')
=> <Schema 1776>

> gf = Factory.new(gs)

> pg = Loader.load_text('grammar', gf, "
> start Point
> Point ::= [Point] \"(\" x:int \",\" y:int \")\"
> ")
=> <Grammar 4756>

> DisplayFormat.print(pg, p)
( 1 , 2 )
```

“Quad” Model


Formatted by
Grammar
Grammar
Instance
of
Grammar
Schema
Formatted
by
Formatted by
Instance of
of
Instance
Schema
Grammar
Formatted
by
Schema
Schema
Instance of

Diagrams

```
class Part
  constraints: SizeConstraints?
  !styles: Style*
end

class Container < Part
  direction: int
  !items: Part*
end

class Text < Part
  string: str
end

class Shape < Part
  !content: Part?
  kind: str?
  location: Point?
  connectors: ConnectorEnd*
end
```

```
class Connector < Part
  label: Text?
  !path: Point*
  !ends: ConnectorEnd*
end

class ConnectorEnd
  arrow: str?
  !label: Text?
  to: Shape / Shape.connectors
  owner: Connector / Connector.ends
end
```

Stencils

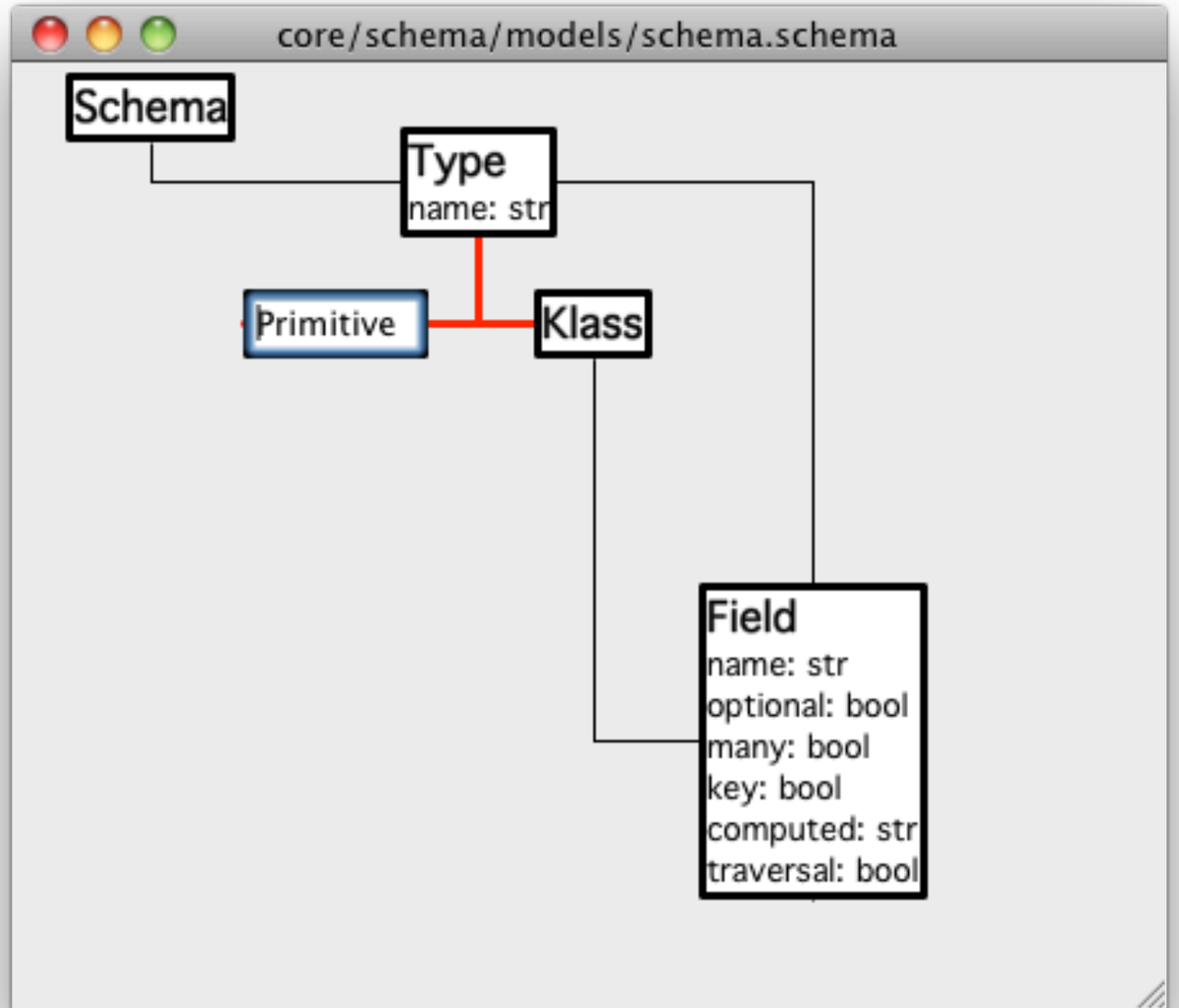
- Template language
- A kind of grammar
  - “parses” models
  - into diagrams
- Stencils are bidirectional
  - (as are grammars)

schema.stencil

```
diagram(schema)

graph [font.size=12] {
  for (class : schema@classes)
   label ClassBox[class]
     box [pen.width=3] {
       vertical {
          text [font.size=16,font.weight=700] class@name
          for (field : class@defined_fields)
            if (field@computed == nil)
             if (field@type is Primitive)
               horizontal {
                 text field@name
                 text ": "
                 text field@type@name } } }
 for (class : schema@classes)
   for (super : class@supers)
     connector [pen.width=3,pen.color=(255,0,0)] (ClassBox[super] --> ClassBox[class])

 for (class : schema@classes)
   for (field : class@defined_fields)
    if (field@computed == nil)
     if (!(field@type is Primitive) & (field@inverse == nil | field@_id < field@inverse@_id)
       connector (ClassBox[field@owner] -- ClassBox[field@type])
}
```

Schema Diagram Editor

Editing schema of schemas

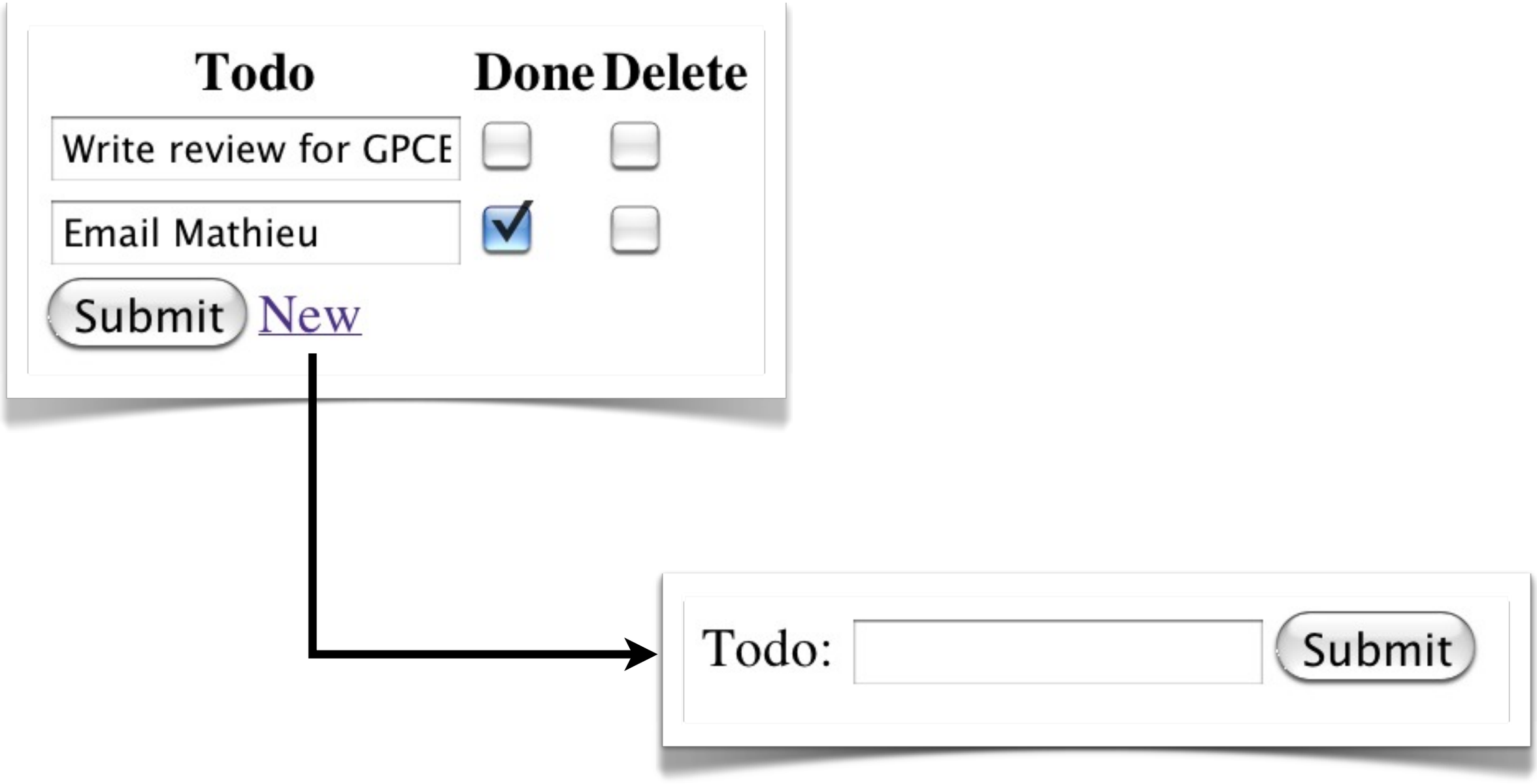


EnsoWeb

```
def index {
  html("Todos") {
    form {
      datatable(root->todos) {
        column("Todo")   { textedit(row->todo); }
        column("Done")   { checkbox(row->done); }
        column("Delete") { delete_checkbox(row); }
      }
      submit("Submit", index());
      navigate("New", new_todo(root->todos, new(Todo)));
    }
  }
}

def new_todo(todos, todo) {
  html("New Todo") {
    form {
      "Todo: " textedit(todo->todo);
      <input type="hidden" name=address(todos) value=address(todo)/>
      submit("Submit", index());
    }
  }
}
```

```
class Todos
  !todos: Todo*
end

class Todo
  owner: Todos / Todos.todos
  todo: str
  done: bool
end
```

Todo App


Todo
Done
Delete
Write review for GPCE
Email Mathieu
Submit
New
Todo:
Submit

Leverage

Leverage

- Web app "business objects" described by schemas
- All models described by schema too
- Hence, Web apps for
  - grammars
  - schemas
  - diagrams
  - web apps
  - ...

Leverage (ctd)

- Grammars provide (de)serialization based on schema
- All models described by schema
- Hence, textual parsing/rendering for
  - grammars
  - schemas
  - diagrams
  - web apps
  - ...

Leverage (ctd)

- Stencils convert any model to a diagram
- Hence, can provide diagram editors for
  - grammars
  - schemas
  - diagrams
  - web apps
  - stencils
  - ...

Ongoing

- Generic operations: diff, print, merge,etc.
- Interpreter composition
  - “Object Algebras”
- Compiling to JS
- Generic debugging (Alex Loh)

Conclusion

- Ensō ~ model-driven programming language/environment/pattern/style/...
- *Models* = what
- *Interpreters* = how
- [2 papers, 4 to go]