Ensō

William Cook, UT Austin
Tijs van der Storm, CWI

Goals

- Eliminate boilerplate
  - 100x or more reduction
- Don't Design Programs
  - Program the design: "run the design"
- Integrated External DSLs
  - Interpreted with partial evaluation
- Consistent, self-hosted system
  - Top half of desktop/server stack

Languages

Data definition
Grammars
Stencils (Diagram/GUIs)
Web UI
Security
Workflow

Strategy

- External DSL
  - not "embedded"
- Interpreters
  - Aspect = Interpreter Composition
- Partial Evaluation
  - interpreter → compiler
- Generic Operations
  - Difference, parsing, etc

"Smalltalk of Modeling"

- Self-Describing
  - Extreme Reuse
  - Extreme Malleability
  - Ruby runtime
- DSL/MDD is the new paradigm
  - Built on top of Objects
  - ... but it is not objects

Next Steps

- Theory
  - Why? How does it fit in PL?
  - Enso details
  - Partial Evaluation
  - Better RPC (loosely related work)
- Demonstrations
  - Diagram Editor
  - Web Applications/Security
  - Distributed Synchronization
  - Read the code :-)

Why DSLs?

Spectrum of Programming

How
implementation

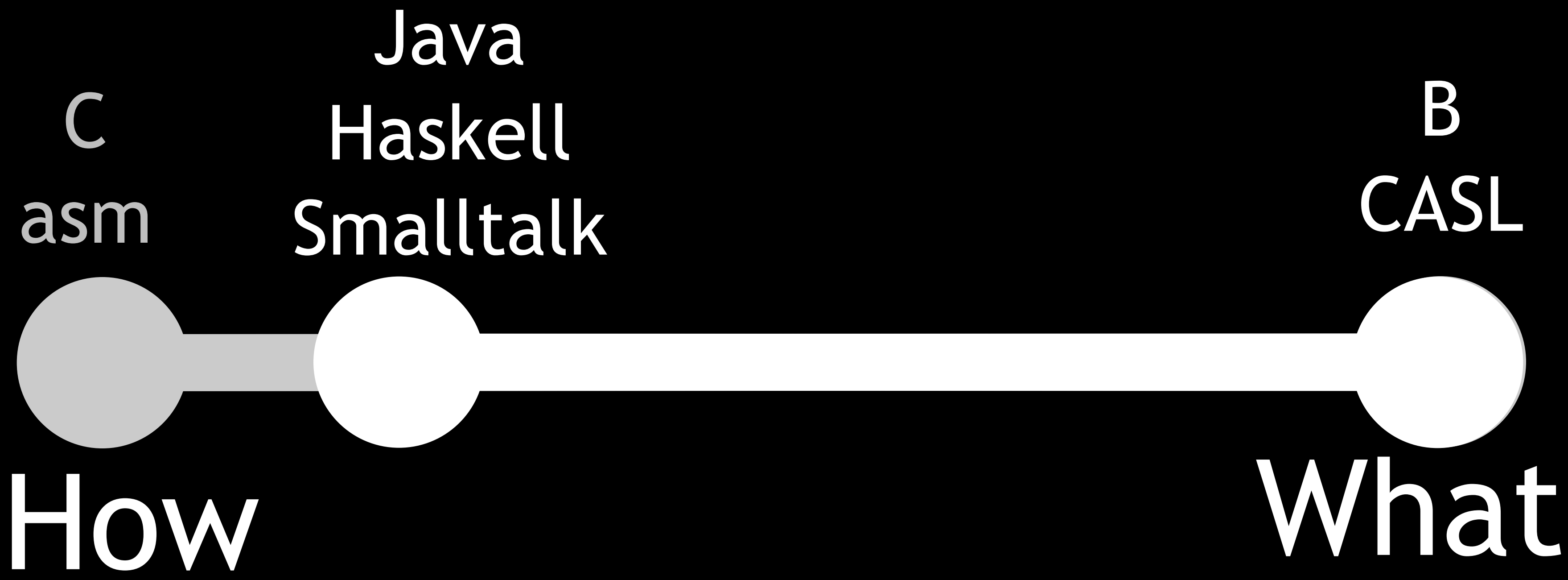
What
Specification

C
asm

B
CASL

How
implementation

What
Specification



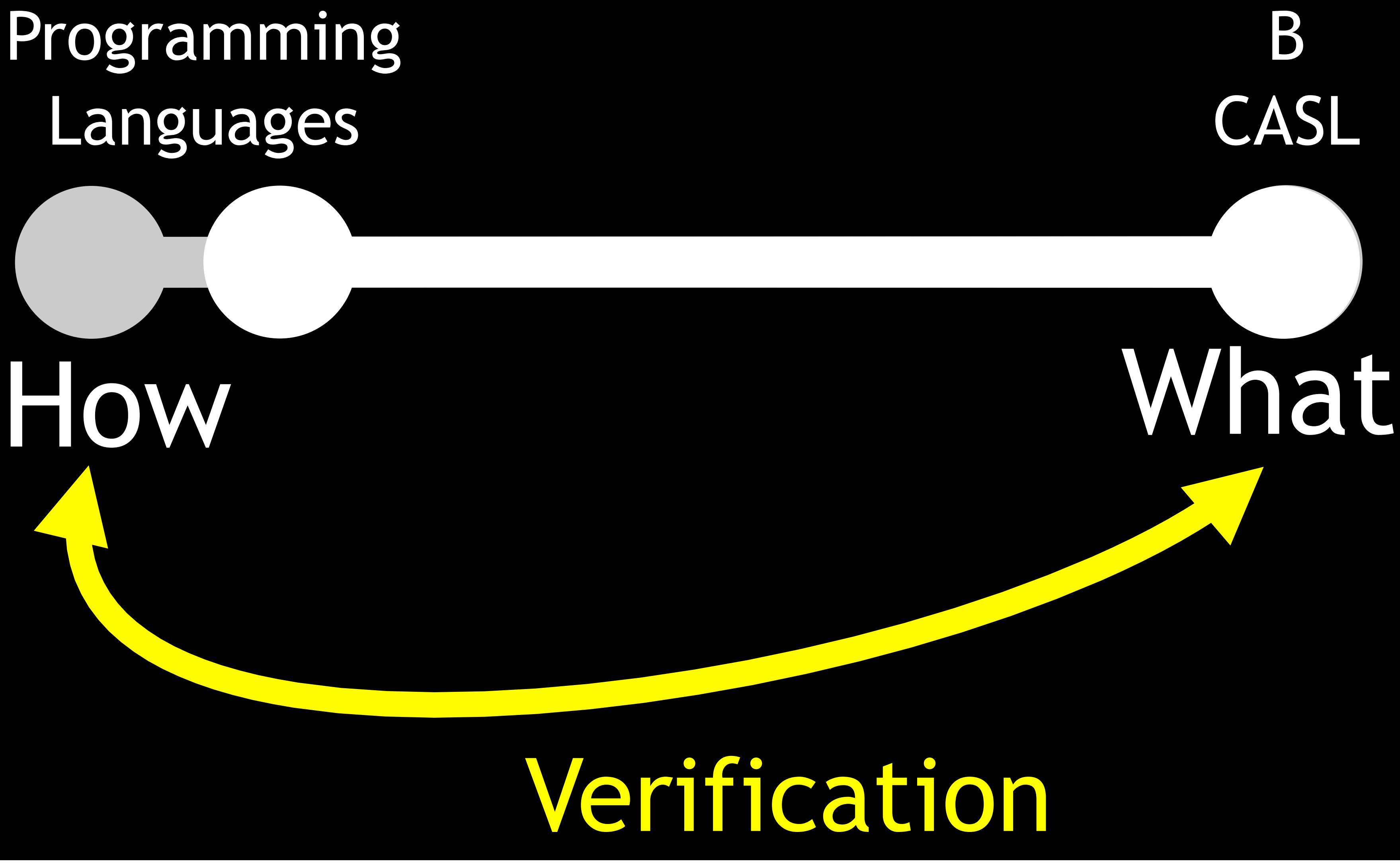
C
asm

Java
Haskell
Smalltalk

B
CASL

How

What



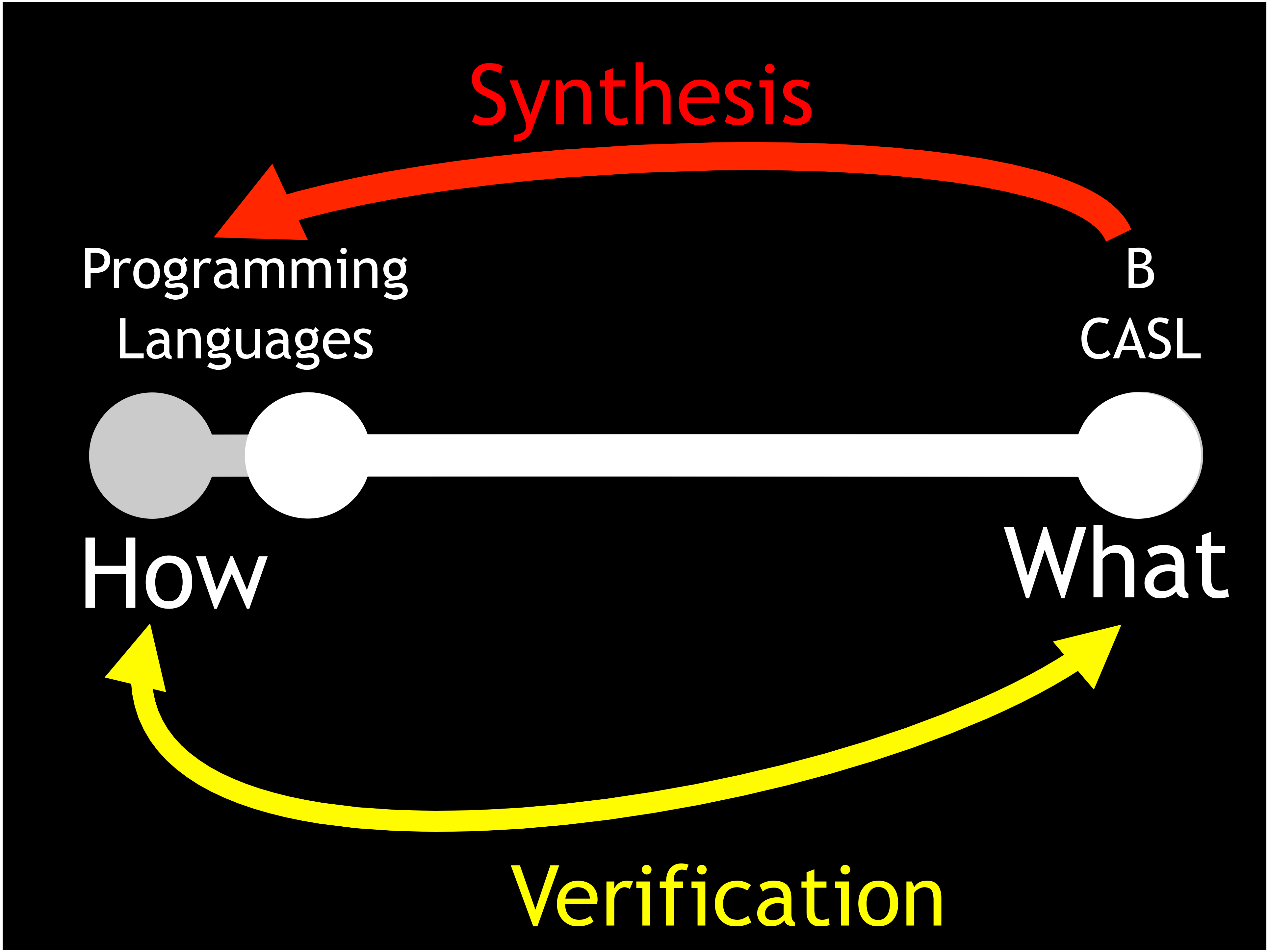
Programming Languages

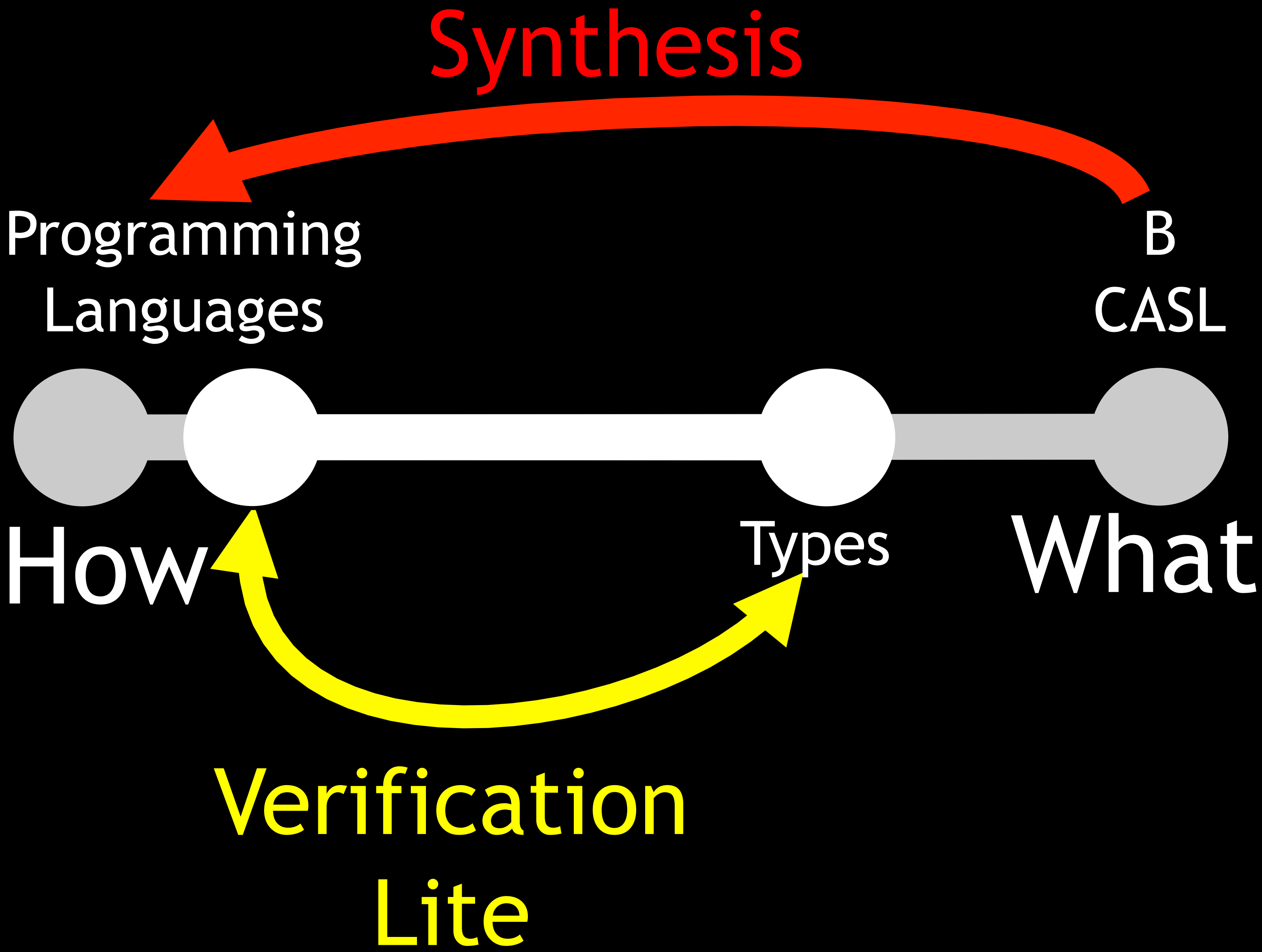
B
CASL

How

What

Verification


Synthesis
Programming Languages
B
CASL
How
What
Verification



Synthesis

Programming
Languages

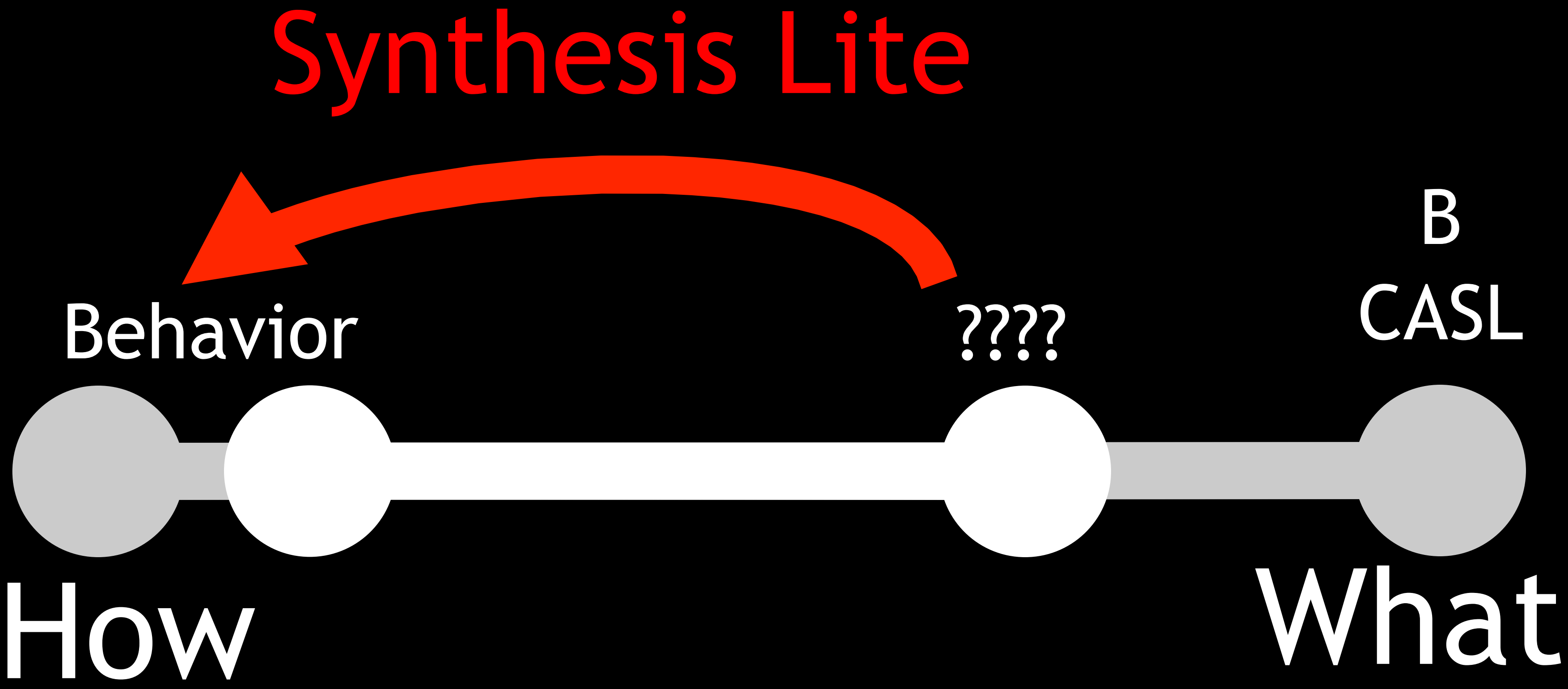
B
CASL

How

Types

What

Verification
Lite



Synthesis Lite

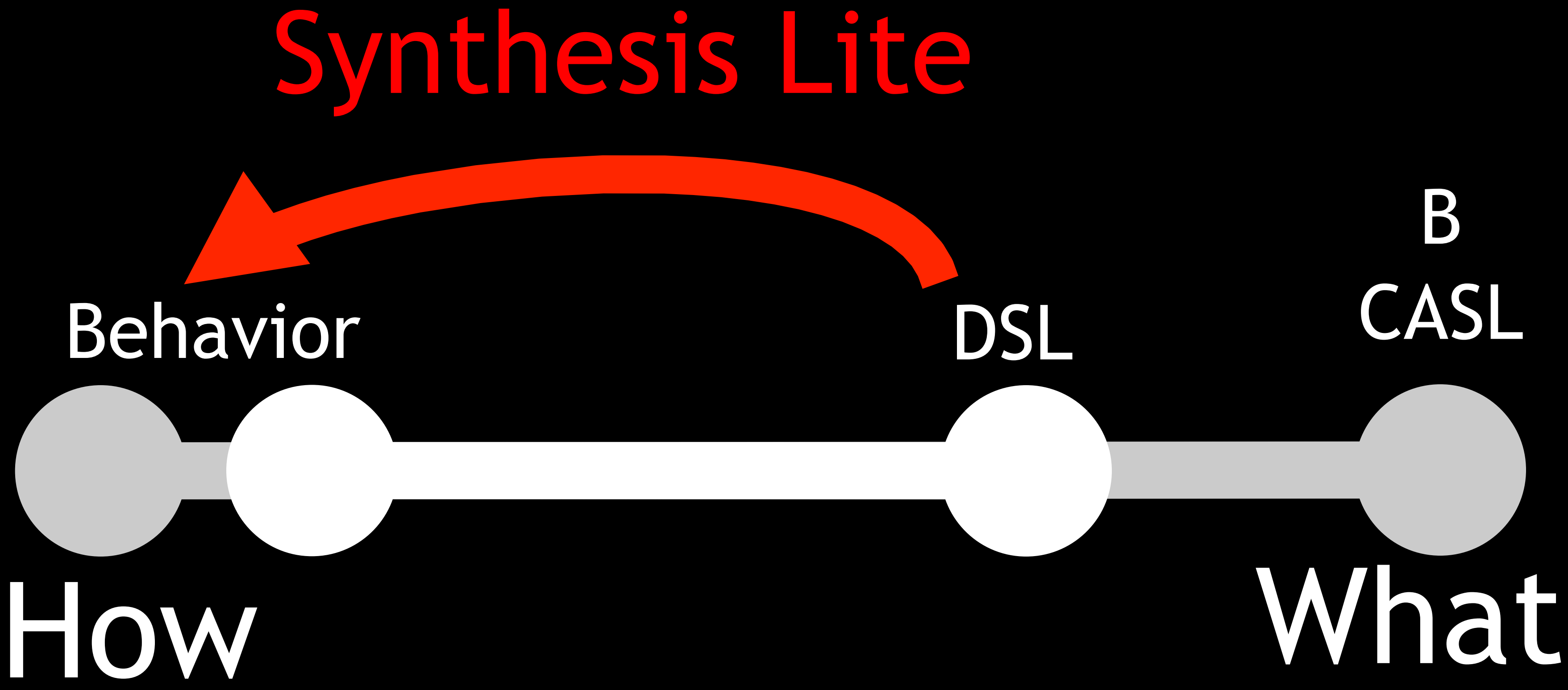
Behavior

????

B
CASL

How

What



Synthesis Lite

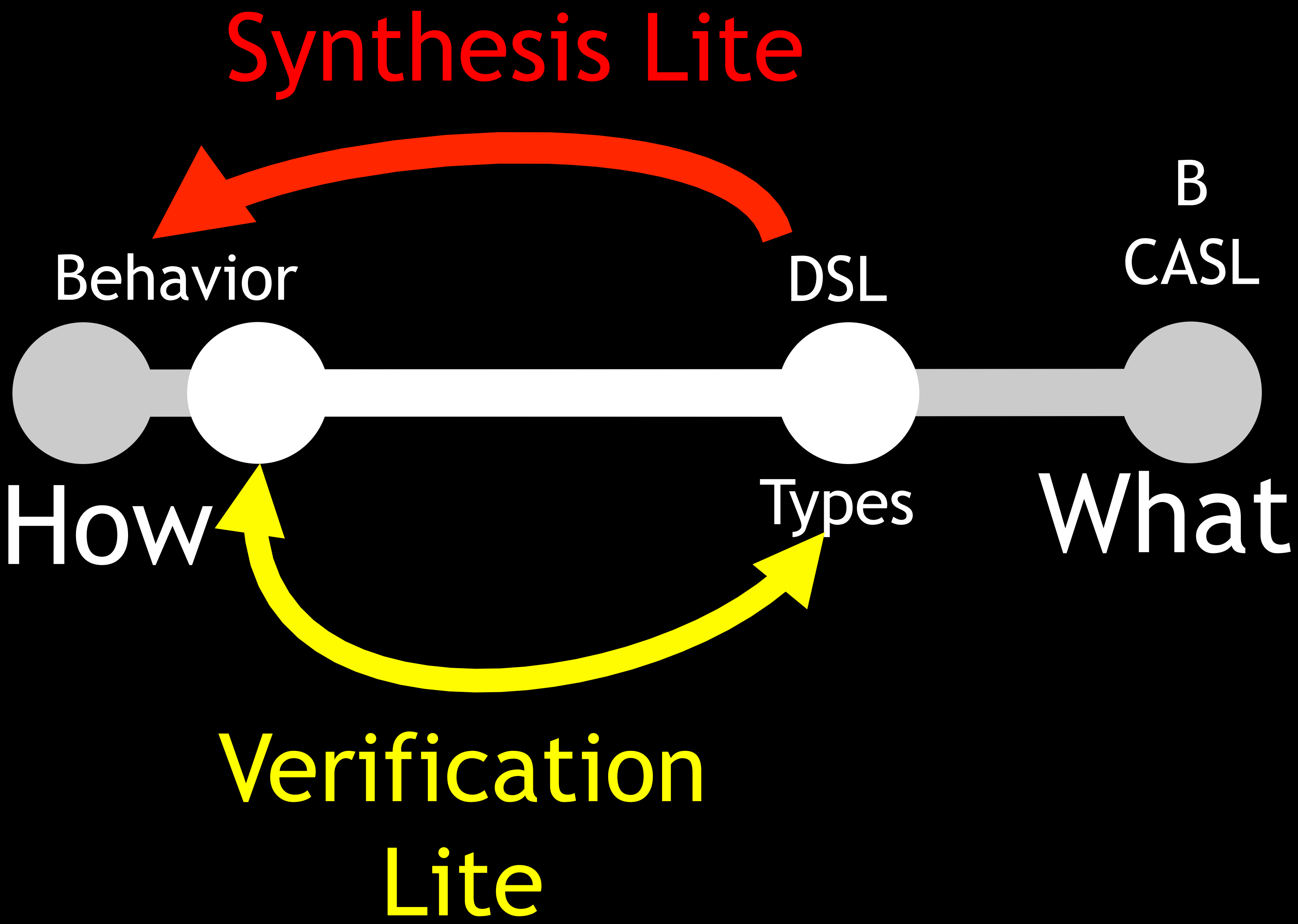
Behavior

DSL

B
CASL

How

What


Synthesis Lite
B
CASL
Behavior
DSL
How
Types
What
Verification
Lite

DSLs

Strategy + Specifics

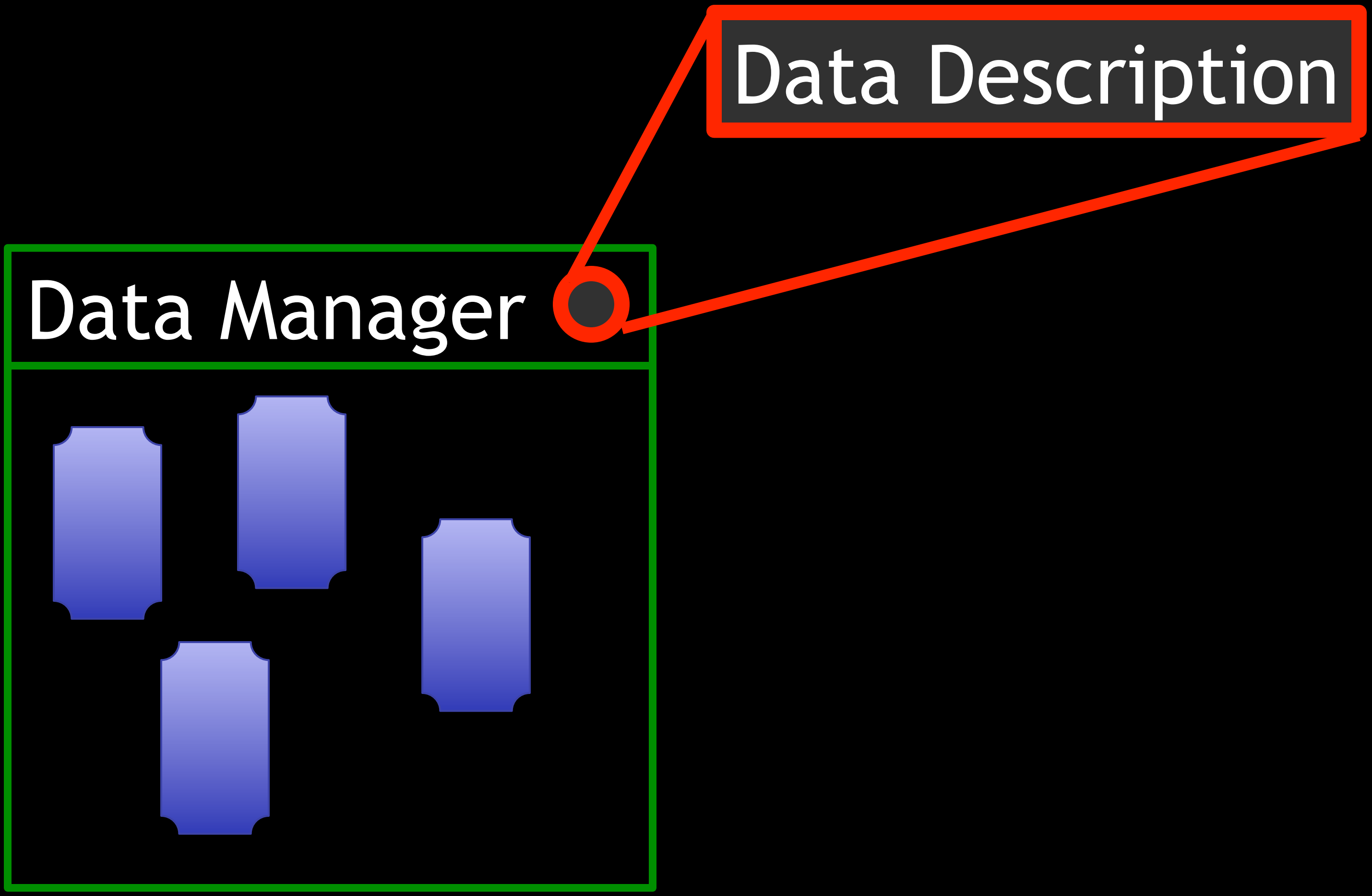
New dimension of modularity

Ensō Data

Managed Data

- Data
  - Don't just program data types/structures
  - Code up your data structuring mechanism
  - In other words, override "dot" in o.f
- Control over data architecture
  - Persistence, access control
  - Bi-directional Relationships
  - Invariants, constraints, derived data

Managed Data



Data Description

Data Manager

Ensō Data

- Graphs with a spanning tree
  - Spanning tree defined by subset of edge labels
- Properties
  - Holistic view of data graphs
  - Bidirectional edges
- Seeking
  - Easy to work with in PL
  - Easy mapping to RDBMS

Point Schema

- A *point* has *x* and *y* integer components

Instance	***Schema***
(3, 4)	class Point x: integer y: integer

- A schema *describes* the structure of data

Description of Schemas

- A *class* has a *name* and a list of *fields*, each of which ha a name and a type

Individual	*Schema*
class Point x: integer y: integer	class Class name: string fields: Field* class Field name: string type: ???

Not explained

Not explained

Schema of Schemas

Individual

```
class Point
    x: int
    y: int
```

Schema

```
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
```

Not explained

Schema Schema

```
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
```

Adding More Metadata

- Inverse fields
  - parent/child, instructor/course, etc
  - automatically maintained
- Field properties
  - key: ensures uniqueness in context
  - traversal: ensures reachability
  - ordering: ordered/unordered collections
- Invariants
  - field.type = field.inverse.owner
- More…
  - Computed fields, etc

Factories

- Factories
  - Manages data described by a schema
- Points

```
factory = Factory.new(PointSchema)
pnt = factory.Point(3, 5)
print pnt.x
```

- Interpreted
  - Virtual object has fields specified by schema
  - Ensures data is valid with respect to schema

Grammars

- A *point* is written as (x, y)

Individual	***Grammar***
(3, 4)	start P P::= [Point] "(" x:int "," y:int ")"

- Notes:
  - Direct reading, no abstract syntax tree
  - Bidirectional: can parse and pretty-print

Schema Grammar

```
start S
S ::= [Schema] types:(P| C)*
P ::= [Primitive] "primitive" name:sym
C ::= [Class] "class" name:sym ("<" super:Class^)?
                 fields:F*
F::= [Field] name:sym ":" type:Type^
            (many:"*” | optional:"?”)?
```

```
primitive int
class Class < Type
   fields: Field*
   super: Type?
```

Bidirectional

- Parsing: matches tokens, generates instances
- Printing: matches instances, generates tokens

```
P ::= [Primitive] "primitive" name:sym
C ::= [Class] "class" name:sym ("<" super:Class^)?
                      fields:F*
```

```
primitive int
class Class < Type
   fields: Field*
   super: Type?
```

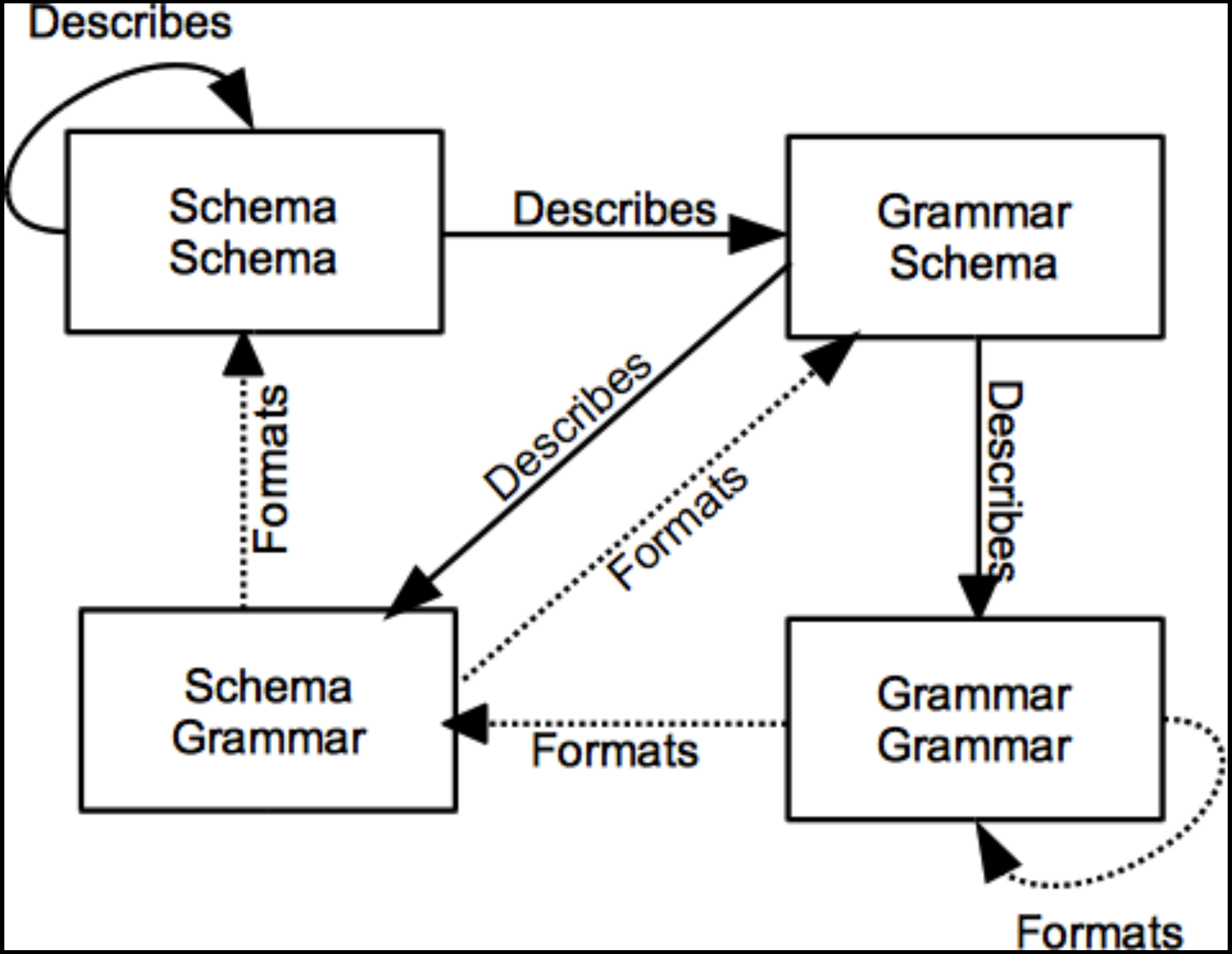
Grammar Grammar

```
G ::= [Grammar] "start" \start:Rule^ 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 ::= [Value] kind:("int" | "str" | "real" | "sym")
    | [Call] rule:Rule^
    | [Ref] name:sym "^"
    | [Lit] value:str
    | [Regular] arg:P "*" @"many = true"
    | [Regular] arg:Pattern "?"  @"optional = true"
    | [Code] "@" code:str
    | "(" A ")"
```

Grammar Schema

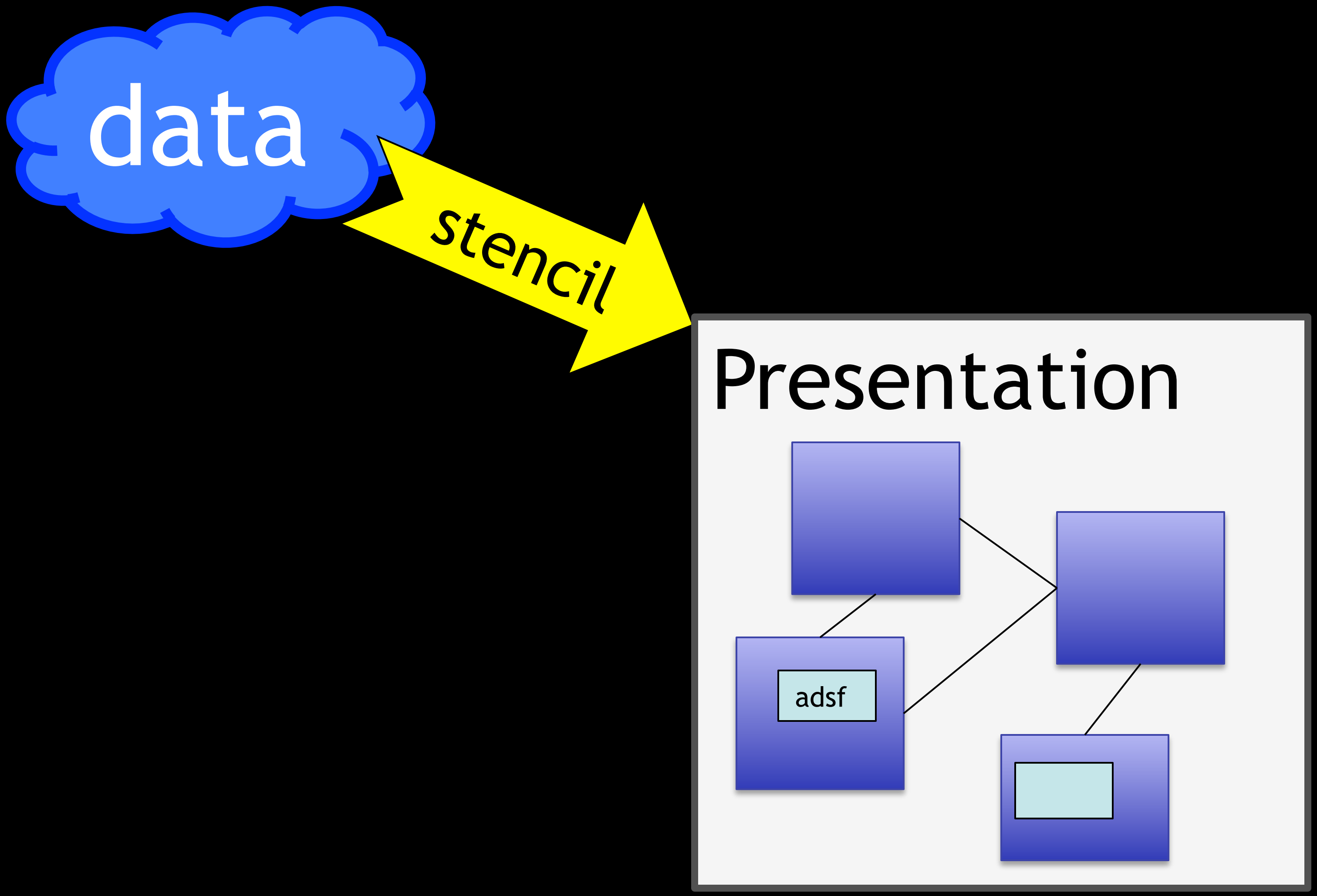
```
class Grammar       start: Rule;    /rules: Rule*
class Exp end
class Rule < Exp:   #name: str;  /arg: Exp
class Alt < Exp:      /alts: Exp+
class Sequence < Exp:  /items: Exp*
class Create < Exp; name: str; /arg: Exp
class Field < Exp;    name: str;   /arg: Exp
class Code < Exp:   code: str
class Value < Exp:   kind: str
class Ref < Exp:      name: str
class Lit < Exp:       value: str
class Call < Exp:     rule: Rule
class Regular < Exp
  /arg: Exp; optional: bool; many: bool; sep: str?
```

Quad Model

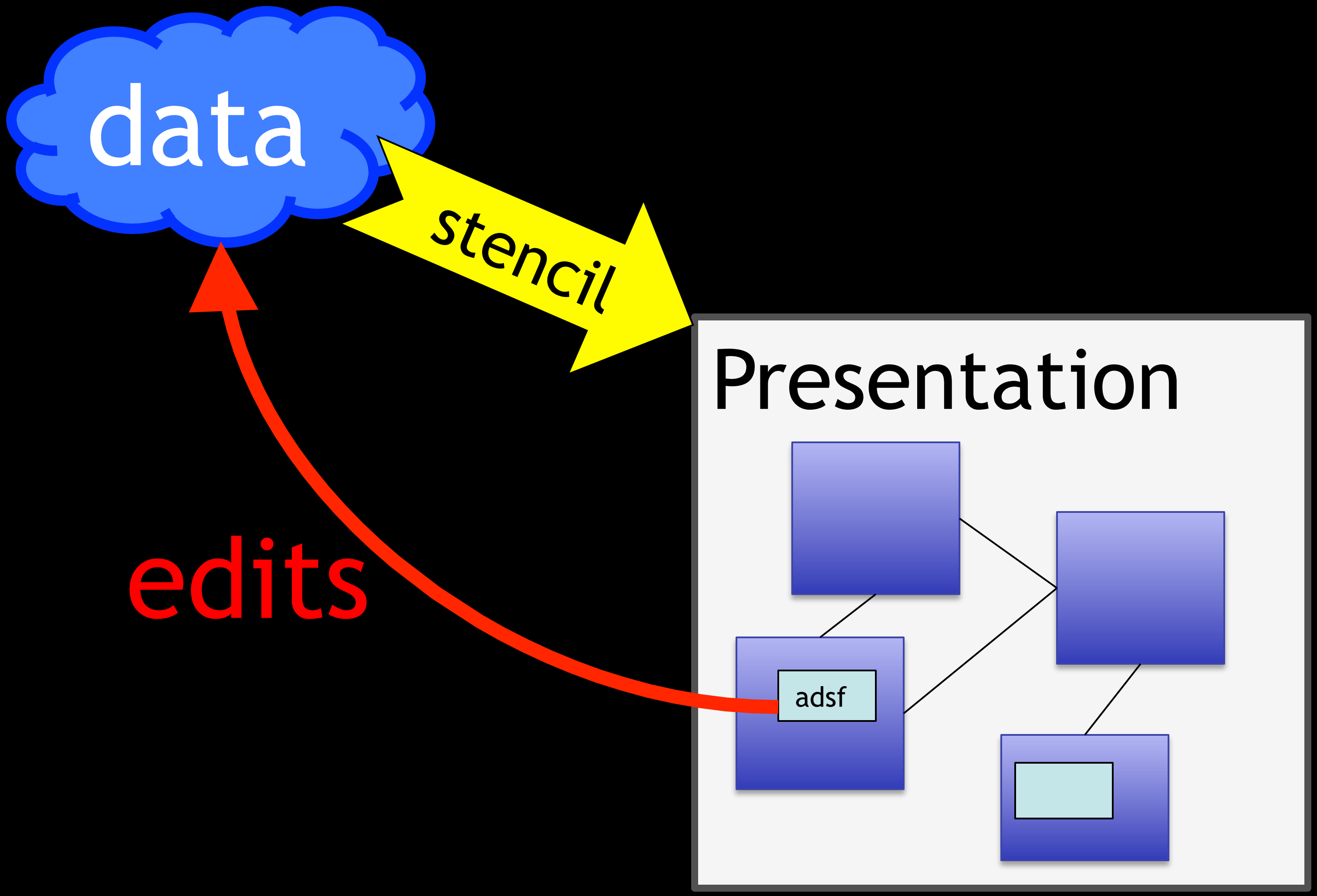

Describes
Schema Schema
Describes
Grammar Schema
Formats
Describes
Formats
Describes
Schema Grammar
Formats
Grammar Grammar
Formats

Stencils

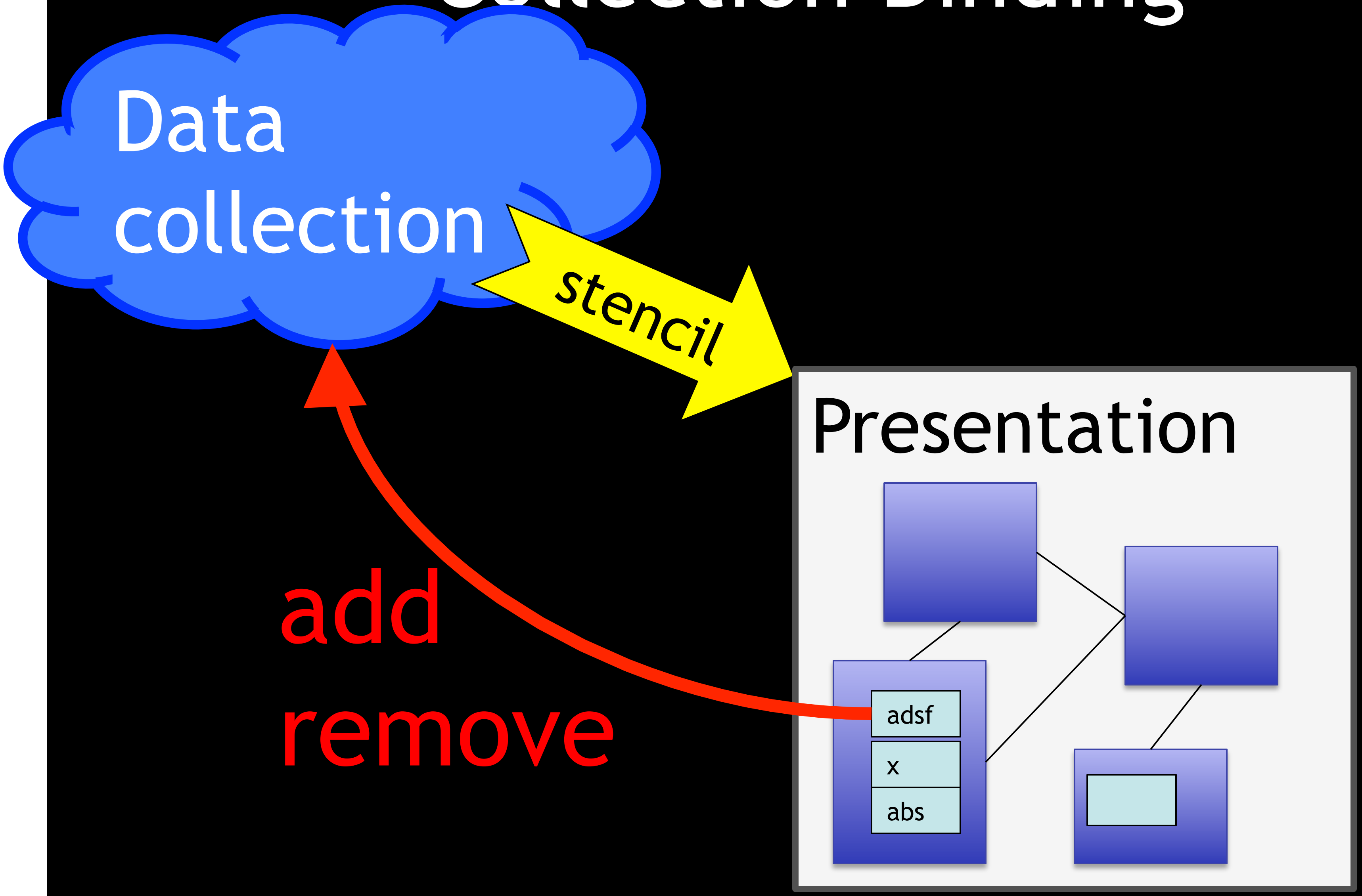
Stencil



(Traditional) Data Binding



Collection Binding



Data collection

stencil

Presentation

add

remove

Web/Security

EnsōWeb

```
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);
        submit("Submit", index());
    }
  }}
```

```
class Todos
  todos: Todo*
end

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

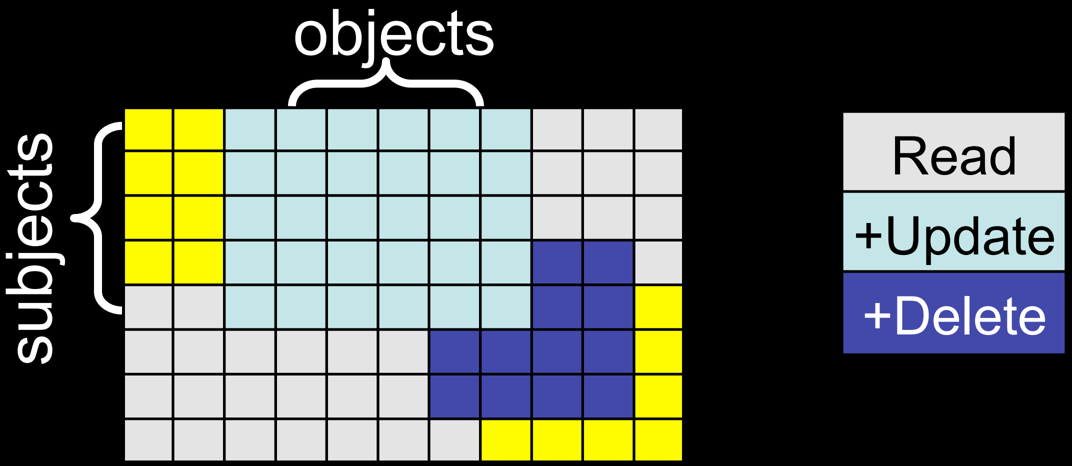
Todo App

Todo	Done	Delete
Write review for GPCE	☐	☐
Email Mathieu	☑	☐

Todo:

Authorization & Access Control

- Authorization
  - Ensure that subjects only perform authorized actions
  - Also: authentication, non-interference, signatures, etc.
- Access Control
  - Subjects are only granted authorized *access* to *objects*
  - Matrix: extensional definition of access at point in time
- Role-Based Access Control
  - Specify access for groups of subjects and objects



EnsōWeb Interpreter

Factory : $\forall S{:}\text{Schema} \rightarrow \text{Data}_S$

Web : pages → Data_S → HTTP → HTML

factory = Factory(**schema**)

response = Web(**pages**, factory, **request**)

Interpreter Wrapping: SQL

Factory : $\forall S$:Schema → Data_S

DB : Data_S → String → Data_S

Web : pages → Data_S → HTTP → HTML

factory = Factory(**schema**)

db = DB(factory, "connection…")

response = Web(**pages**, db, **request**)

Wrapping: Security

Factory : $\forall S$:Schema → Data$_S$
DB : Data$_S$ → String → Data$_S$
Secure : Data$_S$ → Policy → User → Data$_S$
Web : pages → Data$_S$ → HTTP → HTML

factory = Factory(**schema**)
db = DB(factory, "connection…")
sec = Secure(db, **security**, user)
response = Web(**pages**, sec, **request**)

Multiple Interpretations

Factory : $\forall S$:Schema → Data$_S$
Extract: pages → HTTP → Query
FastDB: Data$_S$ → String → Query → Data$_S$
Web : pages → Data$_S$ → HTTP → HTML

factory = Factory(**schema**)
sql = FastDB(factory, "connection…")
query = Extract(**pages**, request)
response = Web(**pages**, sql(query), **request**)

EnsōWeb Concepts

- Composition by wrapping
- Wrappers implement same interface as wrappee
- Clients needn’t know extensions
- Partial evaluation to weave dynamic delegation overhead into code
  - Experiments in last version of Ensō, but not in current version yet

Generic Differences

$\Delta : \text{Schema} \to \text{Schema}$

The delta function takes a data type S and creates a data type for representing "changes to S"

diff: $\forall S{:}\text{Schema} \to (S, S) \to \Delta S$

patch: $\forall S{:}\text{Schema} \to S \times \Delta S \to S$

conflicts: $\forall S{:}\text{Schema} \to (\Delta S, \Delta S) \to [(\Delta S, \Delta S)]$

merge: $\forall S{:}\text{Schema} \to (\Delta S, \Delta S, [(\Delta S, \Delta S)]) \to \Delta S$

EnsōSync (a mini-Unison)

```
def sync(base, s1, s2)
  d1 = diff(base, s1)
  d2 = diff(base, s2)
  confs = conflict(d1, d2)

  r = resolve!(confs)          -- possibly human intervention
  d = merge(d1, d2, r)
  new_base = patch(base, d)
  update1 = diff(s1, new_base)
  update2 = diff(s2, new_base)

  apply(update1)                         -- write to disk
  apply(update2)
end
```

Conclusion

- Executable Specifications
- Interpreters and Composition
- Partial Evaluation

- Collaborate?