Grassroots Ontologies
Context Through Multiplicity

Spencer Breiner\textsuperscript{1} & Eswaran Subrahmanian\textsuperscript{1,2}

spencer.breiner@nist.gov
sub@cmu.edu

\textsuperscript{1}U.S. National Institute of Standards and Technology

\textsuperscript{2}Carnegie Mellon University

January 24, 2018
Grassroots ontologies

How can we build context into our ontologies?
Grassroots ontologies

How can we build context into our ontologies?

How can we build ontologies from a given context?

What does this buy us?

• No need to enforce top-down restrictions.
• No need to modify upper structures to accommodate new contexts.
• Low barriers to entry.

What are the worries?

• Babble
• Rework
Our strategy

• Use category theory to define mappings (functors) between ontologies.

• Build situation-specific ontologies linked together by mappings.

• Aggregate local ontologies over time and across mappings to develop domain-specific libraries.

• Build mixed contexts by inheriting from libraries and integrating with colimits.
Categorical Logic

The main idea:

Every logical ontology can be represented as a category.

Categories support mappings called functors.

We can use functors to map between ontologies.

A secondary idea:

Colimits aggregate diagrams of categories and functors.

Context-specific ontologies define such diagrams.

Colimits can be used to integrate different contexts.
What are categories?

Categories consist of

- *objects* \((X, Y, Z, \ldots)\)
- and *arrows* \((f : X \rightarrow Y, g : Y \rightarrow Z, \ldots)\)

\[
\begin{array}{c}
X \xrightarrow{f} Y \xrightarrow{g} Z
\end{array}
\]

- which can be *composed*

- and may allow additional constructions like *products* \((X \times Y)\).
Example: Schedules

<table>
<thead>
<tr>
<th>Pool Lifeguard Schedule</th>
</tr>
</thead>
</table>
| Alice | M,T,W 12:00-3:00  
| Bob | M, W, F 2:00-5:00  
| | Tu 2:30-5:00  
| Carrie | Th 12:00-5:00  
| | F 12:00-3:00  

We can model this with a simple ontology

- Each person has a set of shifts.
- Each shift has a day, start time and stop time.

\[
\text{Shift} \rightarrow \text{worker} \rightarrow \text{Person} \\
\text{start} \downarrow \text{stop} \downarrow \text{(Time, $\leq$) \rightarrow Day}
\]
Example: Schedules

This schema forms a simple category $S_0$.

We think of objects and arrows as placeholders for sets and functions. An assignment like this is called an *instance*.
Example: Simple Constraints

Axioms and inferences can be encoded through factorization of arrows.

Every shift stops after it starts.

\[
\exists! \text{ (exists unique)} \quad \text{Shift} \rightarrow [t_1 \leq t_2]
\]

\[
\langle \text{start,stop} \rangle \quad \text{Time} \times \text{Time}
\]
Axioms and inferences can be encoded through factorization of arrows.

No person is assigned to overlapping shifts.

\[
\begin{align*}
[S.\text{worker} = S'.\text{worker}] &\rightarrow \neg \neg \neg \neg \neg \neg \neg \neg \rightarrow [(t_2 \leq t_3) \vee (t_4 \leq t_1)] \\
\text{Shift} \times \text{Shift} &\rightarrow \text{Time}^4
\end{align*}
\]
## Theories as categories

<table>
<thead>
<tr>
<th>Logic</th>
<th>Categories</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Object</td>
<td>Task, Person</td>
</tr>
<tr>
<td>Relation</td>
<td>Subobject</td>
<td>$[t_1 \leq t_2] \rightarrow \text{Time}^2$</td>
</tr>
<tr>
<td>(Open) Term</td>
<td>Arrow</td>
<td>$S. \text{worker} : \text{Person}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{worker} : \text{Shift} \rightarrow \text{Person}$</td>
</tr>
<tr>
<td>Constant</td>
<td>Global section</td>
<td>$M : \ast \rightarrow \text{Day}$</td>
</tr>
<tr>
<td>Connective</td>
<td>(Co)limit</td>
<td>$[(t_1 \leq t_2) \cap (t_3 \leq t_0)]$</td>
</tr>
<tr>
<td>$(\cap, \cup, \subseteq, \ldots)$</td>
<td>$\text{Time}^2 [t_3 \leq t_0]$</td>
<td>$[t_1 \leq t_2] \times [t_3 \leq t_0]$</td>
</tr>
</tbody>
</table>
Now suppose that we are in a machine shop, scheduling jobs.

Each job consists of one of these two processes:

**Process A**

- A1: Turn, 5 min.
- A2: Cut, 2 min.
- A3: Sand, 4 min.

**Process B**

- B1: Cut, 4 min.
- B2a: Drill, 3 min.
- B2b: Drill, 3 min.
- B3: Sand, 5 min.
Example: A New Context

A single process schedule has the following form

\[ S_1 := \{ \text{TaskOrder} \} \]

\[ \text{Time} \xleftarrow{\text{start}} \text{Task} \xrightarrow{\text{mach}} \text{Machine} \]

If we think of \text{Machine} \approx \text{Person} and \text{Task} \approx \text{Shift}, then we can view \( S_1 \) as a mixture of \( S_0 \) with the directed graph pattern:

\[ G := \{ \text{Edge} \Rightarrow \text{Node} \} \]
Example: A New Context

A single process schedule has the following form

\[ S_1 := \{ \begin{array}{c}
\text{TaskOrder} \\
\text{Time} \quad \text{Task} \quad \text{Machine}
\end{array} \quad \begin{array}{c}
\text{pred} \quad \text{succ} \\
\text{start} \quad \text{stop} \\
\text{mach}
\end{array} \quad [t_0 \leq t_1] \]

We can embed additional context by adding new constraints:

\[(\text{pred.stop}, \text{succ.start}) \quad \text{Time} \times \text{Time}\]
Functors are formal analogies.

These two scheduling contexts have something in common. We can make this explicit by building functors between them.

- Inclusion functor $F : S_0 \rightarrow S_1$
  
  \( \text{Shift} \mapsto \text{Task}, \text{Person} \mapsto \text{Machine}, \ldots \)

- Projection functor $G : S_1 \rightarrow S_0$.
  
  \( \text{Task} \mapsto \text{Shift}, \text{Machine} \mapsto \text{Person}, \text{TaskOrder} \mapsto \emptyset, \ldots \)

Instances are functors, too:

\[
I : S_0 \rightarrow \text{Sets} \\
\text{Shift} \mapsto \{S_0, \ldots, S_8\}
\]
Syntax/Semantics Duality

Instances pull back along functors.

\[ I : S_1 \rightsquigarrow F^*(I) : S_0 \]
\[ J : S_0 \rightsquigarrow G^*(J) : S_1 \]

Formally, this is just composition of functors:

\[ S_0 \xrightarrow{F} S_1 \xrightarrow{I} \text{Sets} \]

- \( F \) forgets the ordering over tasks. E.g., translate constraints on \( S_0 \)-instances into statements about \( S_1 \)-instances.

- \( G \) encodes the fact that a schedule without order constraints is a special case of one with them.
Sets of plans

Back in the machine shop, we need to worry about many processes, not just one.

We can model this using the slice (meta-)pattern:

$$\mathcal{PS} := \{ \text{TaskOrder} \rightarrow \text{Process} \}$$
Example: Scheduling problems

Having a schedule is a different context from *wanting* one.

Abstracting and extending $\mathcal{PS}$ we get a model for scheduling problems in the machine shop:

$$\mathcal{SP} := \begin{cases} \text{Job} \rightarrow \text{Process} \rightarrow \text{Step} \rightarrow \text{MachineType} \rightarrow \text{Machine} \\ \text{isOrderFor} \downarrow \\ \text{duration} \downarrow \\ \text{StepOrder} \rightarrow \text{Duration} \\ \text{machType} \rightarrow \text{type} \end{cases}$$
Derived concepts

We build bridges between ontologies by identifying their overlap. Here, the overlap is hidden. Tasks and task orderings are a derived concepts in $S\mathcal{P}$:

$$[j.\text{isOrderFor} = s.\text{stepOf}]$$
Now we can identify an explicit overlap between $S_1$ and $\overline{SP}$.

\[ O = \{ \text{TaskOrder} \Rightarrow \text{Task, Resource} \} \]
Building Bridges

Now we can identify an explicit overlap between $S_1$ and $\overline{SP}$.

*Colimit* operations allow us to aggregate the pieces into a common context $C$, respecting the overlap.

\[ O = \{ \text{TaskOrder} \Rightarrow \text{Task, Resource} \} \]
Example: Mixing contexts

By extending $C$, we can specify how different contexts interact.

For example, the difference between the start and stop times in $S_1$ should be the same as the duration of the associated step $\overline{SP}$:
A note on naming

The names we use to describe objects and arrows play *no role* in the mathematical definitions.

This provides tremendous flexibility in managing terminological disputes:

Translate, don’t regulate!

We can still use names to guide our constructions (e.g., Task \(\mapsto\) Task).

Caveat) This means CT provides no guidance in how to resolve naming disputes.
Example: Working with tools

Every tool or formal method presents its own context.

For example, a simple optimization problem might look like this:
Example: Working with tools

We can define a functor $R : \mathcal{OP} \rightarrow \mathcal{SP}$.

For example, the scheduling constraints can be indexed by:

- $\text{Task}^2$ (start $\leq$ stop & correct machine type)
- $\text{TaskOrder}$ (task ordering respected)
- $\text{Task} \times \text{Task}_{\text{Machine}}$ (no machine assigned overlapping tasks)

Hence

$$R : \text{Constraint} \mapsto (\text{Task} + \text{Task} + \text{TaskOrder} + \text{Task} \times \text{Task}_{\text{Machine}}).$$

Pulling back along $R$ translates scheduling problems into optimization problems.
Our vision: Context Everywhere

- Every context has an ontologies.
- Every overlap has a bridge.
- Over time, aggregate, abstract and simplify overlapping domains.
- Develop domain-specific and tool-specific libraries.
- Aggregate libraries for complex contexts, adding context as necessary.
- Translate data and manage workflow through functors.
Why it works

Mathematically:

- Relationship with formal methods (graphs, probability, dynamics, etc.)
- Self-reference (categories & functors form a category)
- Context-relative understanding of syntax & semantics

Pragmatically:

- Logical models are directly tied to applications.
- Iterative construction of bridges and aggregates matches the design process in science and engineering.
Thank you!

Questions?