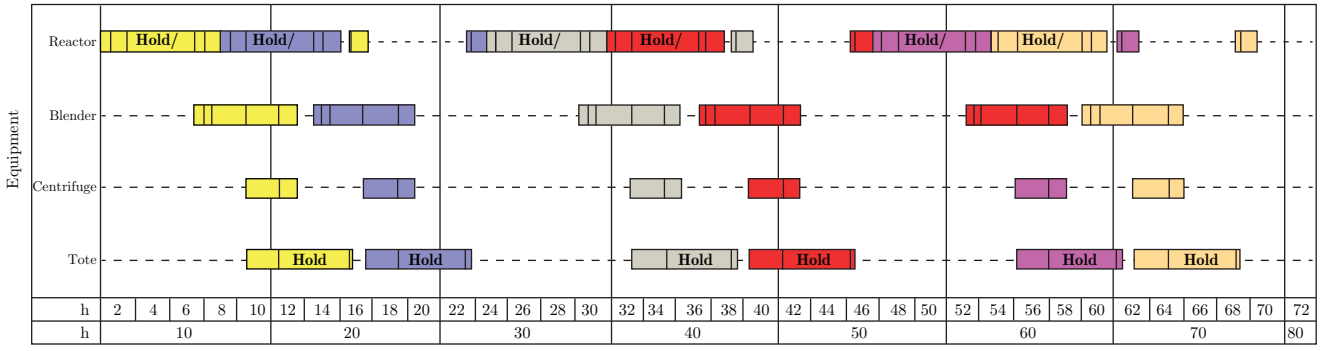
Categories for Planning and Scheduling



Spencer Breiner, Peter Denno, and Eswaran Subrahmanian

1. Introduction

Applied category theory is an ongoing effort to use the abstract machinery of category theory (CT) to organize mathematical knowledge across a wide range of science and engineering [BPS20]. Mostly this requires a change of focus; the same tools designed to handle cohomology and quantum groups fit surprisingly well with more down-to-earth subjects like process planning and error diagnosis [BJS19, BMRPS20].

This paper gives an introduction and case study in the use of CT for data modeling and transformation. The subject of the case study is a standard problem in operations research called *open-shop scheduling*. We make no claim to the originality of our methods, which ultimately derive from the well-developed field of categorical logic [AR94]. Nonetheless, we hope that grounding this machinery in a more pedestrian context will make the ideas easier to digest. The presentation is not self-contained—we won't review standard definitions of functors and natural transformations—but we have tried to minimize our reliance on prior knowledge. We recommend [Spi14] for a more careful development.

Our primary goal is to demonstrate the capabilities of category theory as a formal modeling language. It provides an extremely precise and expressive set of constructions both internally (within the model) and externally (in the meta-theory). The general principles are relevant for both formal modeling languages like OWL, SysML or Modelica as well as generic modeling contexts like programming, database design and user interface. As we will show, CT supports a unique bottom-up methodology, with many small models linked together in a web of functorial relations. The rest of the paper is organized to highlight different features of the approach.

Section 2 introduces *functorial semantics*, the principle that nearly any model (in an informal sense) can be represented as a structure-preserving mapping

$$\textbf{model} : \textbf{Syntax} \longrightarrow \textbf{Semantics}.$$

Syntax specifies the parts of the model and their relationships; the semantics determines how these relationships interact. This approach is quite general, ranging from classical logic to quantum computation, and plays an important role in the theory of programming languages. We consider a more pedestrian example, constructing a simple category $\mathcal{P}$ to represent open-shop scheduling, a standard problem in operations research.

Spencer Breiner is a mathematician in the Information Technology Laboratory at the US National Institute of Standards and Technology. His email address is spencer.breiner@nist.gov.

Peter O. Denno is a computer scientist in the Engineering Laboratory at the US National Institute of Standards and Technology. His email address is peter.denno@nist.gov.

Eswaran Subrahmanian is a research professor in the Engineering Research Accelerator and the Department of Engineering and Public Policy at Carnegie Mellon University. His email address is sub@cmu.edu.

Communicated by Notices *Associate Editor Emilie Purvine.*

DOI: https://doi.org/10.1090/noti2186

	Jobs	j_1	j_2	j_3	j_4
Machines	saw	2 hr	2 hr	2 hr	2 hr
	drill	2 hr	3 hr	0	3 hr
	lathe	2 hr	3 hr	3 hr	0
	mill	2 hr	2 hr	2 hr	3 hr

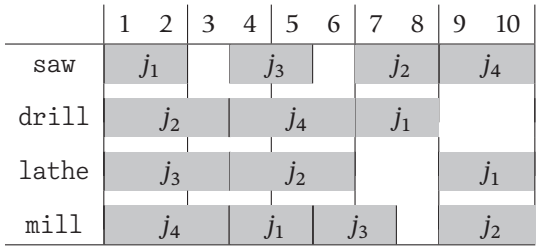


Figure 1. An open-shop scheduling problem and solution involving four jobs and four machines.

The most novel feature of categorical modeling is its extensive use of functorial (composition-preserving) relationships between data models. We illustrate this in Section 3 by defining an open-shop solution category $\mathcal{S}$ along with a functor $F : \mathcal{P} \to \mathcal{S}$. Together with functorial semantics, F induces canonical change-of-base transformations (Δ_F, Σ_F) that can be used to move data from one context to another.

Starting from an extremely limited vocabulary (objects, arrows, composition), CT achieves remarkable expressivity through an extensive collection of design patterns used to model commonly recurring phenomena. In Section 4 we introduce just one of these—a design pattern for indexed families called the *slice* construction—and show how it can be used to generalize the traditional open-shop problem in two different ways.

In Section 5 we extract a scheduling problem on-the-fly from independent but overlapping data sets, demonstrating a very general approach to data integration. We rely on CT's self-referential meta-theory: schemas and functors form a category, so categorical constructions can be used to specify modeling operations.

Section 6 introduces a discussion of optimization and objective functions, which are ignored up until that point. These raise some subtle points regarding the definability of aggregates like n-ary sums and maxima. Here we give a small novel contribution, formally defining an aggregation strategy based on labeled bundles.

Finally, Section 7 puts these pieces together to construct a heuristic solution for $\mathcal{P}$. We'll see that an algorithmic solution to the problem (almost!) defines a second functor $A : \mathcal{S} \to \mathcal{P}$, mapping counter to the problem formulation F.

2. Open Shop Problems

In a typical scheduling problem the goal is to arrange the processing of $\mathtt{J}$ *jobs* across $\mathtt{M}$ *machines*, given that job j must be processed on machine m for time (duration) $\tau_{jm} \geq 0$. This is usually represented as a $\mathtt{J}\times\mathtt{M}$ matrix of non-negative numbers, as in Figure 1. When $\tau_{jm} = 0$, this indicates "no processing," and we refer to a pair jm with $\tau_{jm} > 0$ as a *task*. There are many variants, corresponding to different underlying models of the factory floor. In this section we introduce the simplest, *open-shop scheduling*, and use it to motivate the main ideas of the categorical data model.

In an open-shop problem, tasks can be ordered however we like, subject to two no-overlap constraints. Thus a *valid schedule* is defined by an assignment of start and stop times $s_{jm} \leq t_{jm}$, with $\tau_{jm} = t_{jm} - s_{jm}$ and

$$\begin{aligned} &\text{(a)} \quad \forall j \neq j' \; \forall m \, \big(t_{jm} \leq s_{j'm}\big) \vee \big(t_{j'm} \leq s_{jm}\big), \\ &\text{(b)} \quad \forall j \; \forall m \neq m' \, \big(t_{jm} \leq s_{jm'}\big) \vee \big(t_{jm'} \leq s_{jm}\big). \end{aligned} \tag{1}$$

The first condition says that a machine cannot work on two jobs at once; the second that each job is processed on one machine at a time. When presented as a scheduling chart (Figure 1) the no-overlap conditions are easy to verify by looking at rows and block labels, respectively.

2.1. Schemas & instances.

These definitions can be summarized as a pair of structured categories, called *schemas*, together with a (structure-preserving) functor $F : \mathcal{P} \to \mathcal{S}$. In this section we discuss the problem schema $\mathcal{P}$, which is easy to specify, but requires some unpacking:

$$\mathcal{P} \coloneqq \left\langle \; \mathtt{J} \times \mathtt{M} \xrightarrow{\tau} \mathtt{R} \; \right\rangle. \tag{2}$$

Section 3 will cover the solution schema $\mathcal{S}$ and the relationship F.

Like any category, $\mathcal{P}$ contains objects and arrows. Some of these are shown explicitly in the declaration, while others can be inferred from context. Here there are four objects ($\mathtt{R}$, $\mathtt{J}$, $\mathtt{M}$, $\mathtt{J}\times\mathtt{M}$) and seven arrows ($\tau$, two product projections, four identities).

We can think of the objects and arrows in $\mathcal{P}$ as variables ranging over sets and functions. An *instance* of a schema is an assignment of concrete sets and functions to these variables; the set of instances on $\mathcal{P}$ is denoted $\widehat{\mathcal{P}}$. For example, the specific open-shop problem shown in Figure 1 corresponds to an instance $P \in \widehat{\mathcal{P}}$ with

$$\begin{aligned} P(\mathtt{J}) &= \{j_1, j_2, j_3, j_4\}, \\ P(\mathtt{M}) &= \{\mathtt{saw}, \mathtt{drill}, \mathtt{lathe}, \mathtt{mill}\}. \end{aligned}$$

We refer to $\{j_1, j_2, j_3, j_4\}$ as the *interpretation* of $\mathtt{J}$ in P.

Similarly, $P(\tau)$ is interpreted by the 4×4 matrix shown in Figure 1. In categorical terms, all this data can be encoded as a functor $P : \mathcal{P} \to \mathbf{Set}$, our first example of *functorial semantics*. Moreover, as we will see in the rest of the section, P is structure-preserving in two different ways.

2.2. **Types.** Some schema elements, which we call *types*, do not act like variables. In the open-shop problem, duration is always a positive number, so R should behave like a constant: $P(\mathtt{R}) \equiv \mathbb{R}^+$. To formalize this we introduce a single subcategory **Type** shared by all schemas, and with the same interpretation in all instances. This represents a "standard library" of common mathematical structures. In practice, **Type** may refer to a fragment of the underlying programming language (e.g., Java in [SSVW17]), and we refer to its arrows as *computations*.

Here we were careful to distinguish the type R from its semantics $\mathbb{R}^+$, but from now on we will freely use standard set-theoretic notation like $\mathbb{N}$ and $\mathbb{R}^+$ as well as familiar algebraic and order-theoretic structures $(+, \leq)$ in our schemas directly, with the understanding that these are included in **Type**.

The fixed interpretation of the types is given by a functor $T_0 : \mathbf{Type} \to \mathbf{Set}$. "Sharing" of types means that every schema comes equipped with an inclusion functor $T_{\mathcal{P}} : \mathbf{Type} \to \mathcal{P}$, and an instance P preserves types if $T_0 = P \circ T_{\mathcal{P}}$. Later on we will need to add free variables to our types (analogous to polynomial rings) to represent unknown information, and in this more general situation the equation above is replaced by a natural transformation $\eta_P : T_0 \Rightarrow P \circ T_{\mathcal{P}}$.

2.3. **Constructions.** The interpretation of $P(\mathtt{J} \times \mathtt{M})$ is neither predetermined (like R) nor freely chosen (like J and M). Instead we compute it recursively, a restriction already implicit in the matrix representation in Figure 1,

$$\underset{\text{in } \mathcal{P}}{\uparrow} P(\mathtt{J} \times \mathtt{M}) \quad := \quad \underset{\text{in } \mathbf{Set}}{\uparrow} P(\mathtt{J}) \times P(\mathtt{M}).$$

We say that P *preserves* or *respects* the product.

We will refer to $\times$ as a *constructor* while J and M are the *arguments*. Applying constructor to arguments yields a *construction* $\mathtt{J} \times \mathtt{M}$. Similarly, an abstract product projection is a constructor, though its arguments $p_1 = p_{1,\mathtt{J},\mathtt{M}}$ are almost always implicit from the context. We will also need the pairing operator $\langle -, - \rangle$, which is used to construct elements of the product (e.g., equation (4) in Section 3).

Another important constructor is the *monic* arrow or *subobject*, indicated by a tailed arrow $\rightarrowtail$. This is a bit different from the product constructor, as it produces neither objects nor arrows. Instead, it produces new equations.

Formally, an arrow μ in any category is monic if $\mu \circ f = \mu \circ g \implies f = g$. If we assert that some arrow in a schema is monic, this acts like a constructor: the input is the antecedent and the output is the conclusion. As we will see in the next section, subobjects play a big role in the categorical approach to logic.

We will introduce a few other constructions as we go, notably *coproducts*, *pullbacks* and *pushouts*. Everything covered here can be formalized in terms of *limit/colimit sketches* [AR94].[1]

Generally speaking, we will assume that any constructions used in a schema are preserved, as they specify the intended interpretation. In an instance, a schema's constructions can be explicitly computed and checked in **Set**. For a functor into another schema, preservation of structure encodes a family of proofs demonstrating that the axioms in the domain (e.g., monicity) are satisfied by their interpretations in the target. The latter will be important in Sections 4 and 7.

The use of constructors makes schema development and interpretation an inherently iterative process. Before we can introduce the function variable τ, we must first construct its domain $\mathtt{J} \times \mathtt{M}$; this itself depends on the prior introduction of variables J and M. Similarly, for an instance P it makes no sense to define the function $P(\tau)$ unless one has already chosen the sets $P(\mathtt{J})$ and $P(\mathtt{M})$ and calculated their product.

3. Schedules & Solutions

3.1. **Axiomatics.** To define an open-shop schedule we need start and stop times for each task:

$$\mathcal{S} := \left\langle \mathtt{J} \times \mathtt{M} \underset{t}{\overset{s}{\rightrightarrows}} \mathbb{R}^+ \mid s \leq t, \text{ no-overlap} \right\rangle.$$

This schema looks similar to $\mathcal{P}$, but expressing the scheduling constraints requires a bit more sophistication. We rely on a general translation from first-order logic that turns formulas and proofs into objects and arrows [AR94]. The elements of the diagrammatic axioms should all be viewed as members of $\mathcal{S}$ (or **Type**, if that is more appropriate).

To a first approximation, every formula φ defines an object $[\![\varphi]\!]$ and every inference creates an arrow $[\![\varphi]\!] \to [\![\psi]\!]$. In particular, an axiom usually asserts the existence of a particular arrow. These inferences can be composed to form larger proofs. To see how this works we will look at an easy example, the ordering assumption. Unfortunately the no-overlap axioms would take us a bit too far afield, but we note that the disjunctions in (1) can be constructed using the coproduct and epi-mono factorization, a categorical generalization of direct image [AR94].

Consider the entailment $(x_1 \leq x_2) \vdash (x_2 - x_1 \geq 0)$, with $x_i : \mathbb{R}$. This corresponds to an arrow q, fitting into a larger diagram in **Type**:

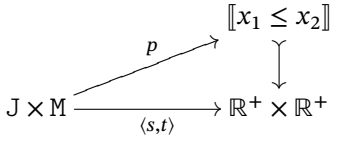
$$\begin{array}{ccc} [\![x_1 \leq x_2]\!] & \xrightarrow{\quad q \quad} & [\![x_0 \geq 0]\!] = \mathbb{R}^+ \\ \downarrow & & \downarrow \\ \mathbb{R} \times \mathbb{R} & \xrightarrow[[\![x_0 := x_2 - x_1]\!]]{} & \mathbb{R} \end{array} \tag{3}$$

[1] *The universes described in Section 6 are more sophisticated, but we are assuming the presence of a universe in* **Type**, *rather than constructing one from scratch.*

Every formula comes equipped with a (usually anonymous) monic embedding into a *context object* (e.g., $\mathbb{R}^n$ for an n-variable formula). Formally these represent tautological formulas like $x_0 = x_0$.

Logical *terms* are built up from variables and function application. Terms define arrows between contexts (bottom row), which act on formulas by substitution. Here, subtraction transforms a one-place formula $x_0 \geq 0$ into the two-place formula $x_2 - x_1 \geq 0$, our desired consequent. Finally, q itself explicitly witnesses the specified entailment, which we can think of either as a true fact about **Set** (such q exists and is unique) or as an axiom in **Type**.

Similarly, each schema axiom asserts the existence of some arrow in the schema, with domain and codomain computed from the axiom's antecedent and consequent. Here, the ordering axiom has a trivial antecedent ($s_{jm} \leq t_{jm}$ for *all* j, m), so the asserted arrow p maps directly out of the context object:

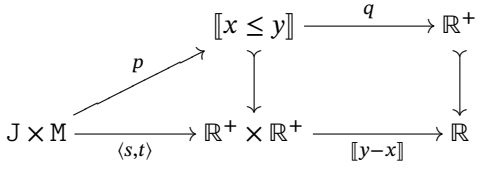


(4)

Similarly, the no-overlap axioms are formulated as a pair of arrows out of $\mathtt{J} \times \mathtt{J} \times \mathtt{M}$ and $\mathtt{J} \times \mathtt{M} \times \mathtt{M}$, respectively.

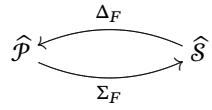
For the functor $F : \mathcal{P} \to \mathcal{S}$ we must send each object and arrow $\mathcal{P}$ to an interpretation in $\mathcal{S}$, while preserving composition and whatever other structure is present in $\mathcal{P}$. Informally, shared naming often indicates common ground between two schemas, but officially different schemas have separate name-spaces. In practice, we eliminate boilerplate by assuming that names are preserved unless otherwise indicated. Here, $\mathtt{J}$, $\mathtt{M}$ and $\mathbb{R}^+$ are all preserved by F, as well as the product object and its projections.

When a domain element does not appear in the target we must construct a definition using whatever structure is available. Here the duration map $\tau \in \mathcal{P}$ does not occur in $\mathcal{S}$ but we can define it as the difference between t and s. Categorically, this corresponds to concatenating diagrams (3) and (4):



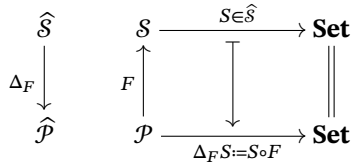
We have to be a bit careful; the naïve interpretation $\tau \mapsto t - s$ corresponds to the bottom row of the diagram, but this has the wrong target: $\mathbb{R} \neq \mathbb{R}^+$. Instead we should use the upper composite $F(\tau) \coloneqq q \circ p$, building the axioms and inferences that justify the interpretation directly into its definition.

3.2. **Change of base.** Any structure-preserving functor $F : \mathcal{P} \to \mathcal{S}$ induces two opposed mappings on instance categories:



Δ_F, the *inverse image*, defines a fundamental duality between syntax and semantics that will inform the rest of our story. Σ_F, the *direct image*, is a bit more technical; it provides a principal means of representing unknown data, using methods conceptually similar to polynomials rings.

Δ_F is easy to define: we just compose:



This is often referred to as a syntax/semantics *duality* because the syntactic functor F induces a semantic mapping Δ_F in the opposite direction. In this case, Δ_F sends a schedule S to the problem P that it solves.

This duality can be elaborated by characterizing dual classes of syntactic and semantics functors, swapping injectivity properties for surjectivity properties. F is an inclusion, injective on objects and arrows; dually, Δ_F is surjective on objects. Every problem has *some* valid solution, since doing the tasks one at a time satisfies no-overlap.

The functorial properties involved are a good deal more complicated than injectivity and surjectivity, and we make no attempt at a systematic study. However, we will note these dualities when they arise. A detailed examination of these issues can be found in [Mak88] (cf. "conceptual completeness").

The second change-of-base functor, Σ_F, is called the *direct image*. It can be defined as a left adjoint to Δ_F, though justifying its existence is non-trivial, and the argument depends on which constructors are used in the schemas [AR94]. The main thing to take away is that Σ_F allows us to push data forward along a functor. It fills in any missing data on the target with placeholders called *labeled nulls*, which act like free variables within **Type**.

More concretely, any element of the instance $x \in PX$ (for some $X \in \mathcal{P}$) generates a corresponding image $F(x) \in (\Sigma_F P)(FX)$.[2] Consequently, whatever jobs and machines are in P will also show up in $\Sigma_F(P)$. The direct image also maps equations so that, e.g., the arrow assignment $P(\tau)(j_1, \mathtt{saw}) = 2$ has a corresponding equation in $\Sigma_F P$.

[2] *The mapping* $x \mapsto F(x)$ *defines a natural transformation* $\eta : \mathrm{id}_{\mathcal{P}} \Rightarrow \Sigma_F.\Delta_F$ *called the* unit *of the adjunction* $\Sigma_F \dashv \Delta_F$. *In Section 5 we will also make use of the dual* counit *transformation* $\epsilon : \Delta_F.\Sigma_F \Rightarrow \mathrm{id}$.

However, this image data is usually incomplete for the target schema, and we must apply a completion operation to obtain a well-defined $\mathcal{S}$-instance. Here we inherit the jobs and machines from P, but have no way to infer the start and stop times s and t. Nonetheless, we are required to provide a function

$$(\Sigma_F P)(s) : (\Sigma_F P)(\mathtt{J} \times \mathtt{M}) \longrightarrow (\Sigma_F P)(\mathbb{R}^+).$$

Rather than select an existing value arbitrarily, Σ_F modifies the target set to include free variables for unknown values. In our example, $(\Sigma_F P)(\mathbb{R}^+)$ is a set of polynomials[3] in 32 unknowns, with two variables s_{jm}, t_{jm} for each of the sixteen tasks in P. However, this space of polynomials is quotiented by the sixteen equations defining $P(\tau)$, leaving us with

$$(\Sigma_F P)(\mathbb{R}^+) = \mathbb{R}^+[s_{jm}, t_{jm}] \Big/ (t_{jm} - s_{jm} = \tau_{jm}).$$

The new elements introduced by Σ-operations are called *Skolem variables* or *labeled nulls*, and their presence distinguishes true solutions from templates. S is called a *ground* instance when it does not contain any missing information [SSVW17]. Here the schemas are quite simple, but in general the interplay between adding new elements and adding new equations can be quite intricate. In practice, Σ_F can be computed using "the chase," an algorithm for testing and enforcing database dependencies [FL19].

Since F and P preserve $\mathbb{R}^+$, while $\Sigma_F P$ sends it to a polynomial ring, it is clear that $P \neq (\Sigma_F P) \circ F$. In other words, $\Sigma_F P$ is not a solution to P. Instead, we should think of $\Sigma_F P$ as a solution *template*, with missing information represented by free variables and interconstraints encoded in the quotient.

To "fill in" the template, we evaluated the polynomials at specified values, defining a canonical natural transformation $\Sigma_F P \Rightarrow S$ for any ground instance $S \in \widehat{\mathcal{S}}$. More generally, we can also represent partial solutions as quotients of $\Sigma_F P$, and these automatically keep track of interconstraints so that, e.g., t_{jm} is assigned as soon as s_{jm} is known (or vice versa). Preservation of types ensures that we don't quotient too far (e.g., 0=1), although this can be challenging to verify in practice and is likely undecidable in general.

4. Variations on a Theme

In this section we modify our schemas to generalize the classical open shop. The categorical data model allows us to express these variations independently, but link them with functors that express their interrelationships. Here we consider two variants, allowing for more flexible notions of tasks and jobs, respectively. Both have a common flavor, relying on a categorical operation called the slice construction which represents indexed sets and functions as commutative triangles.

The basic observation is that a function $p : Y \to X$ can be viewed as an indexed family of sets $Y_x = p^{-1}(x)$. To emphasize this perspective we sometimes call Y a *bundle* over the *base* X; p is the *projection* (often implicit from context), and the sets Y_x are its *fibers*.

In what follows, we use bundles to generalize two different elements of the open shop. In Section 4.1 we modify the job descriptions, introducing a bundle of tasks indexed by $\mathtt{J} \times \mathtt{M}$. In Section 4.2 we bundle machines over their capabilities, allowing duplicate functionality and some parallel processing.

4.1. Duplicate tasks.

In the traditional open-shop model we assume that each job visits each machine once (at most), but sometimes this can be unnecessarily restrictive. If job j involves two independent operations o_1 and o_2 on machine m, then splitting these up can provide additional flexibility in scheduling.

Somewhat surprisingly, the additional flexibility makes our schema simpler, at least conceptually if not visually:

$$\mathcal{P}' := \left\langle \begin{array}{ccc} \mathtt{T} & \xrightarrow{\tau'} & \mathbb{R}^+ \\ {\scriptstyle j}\downarrow & \searrow^{m} & \\ \mathtt{J} & & \mathtt{M} \end{array} \right\rangle.$$

We now have an arbitrary set of tasks $\mathtt{T}$ with explicit projections to $\mathtt{J}$ and $\mathtt{M}$. These were implicit in the earlier schema, but all we have done formally is to *remove* the assumption that these projections form a product structure.

There is an obvious functor $G : \mathcal{P}' \to \mathcal{P}$ that sends $\mathtt{T} \mapsto \mathtt{J} \times \mathtt{M}$ (and $j, m \mapsto p_1, p_2$). Logically speaking, this corresponds to the addition of an axiom $\mathtt{T} \cong \mathtt{J} \times \mathtt{M}$. This is a quotient operation on schemas, a surjectivity property, and dually Δ_G is a full and faithful inclusion exhibiting $\mathcal{P}$-instances as a subclass of $\mathcal{P}'$-problems.

Definitions in $\mathcal{P}$ allow for additional precision. In Section 2 we defined tasks to exclude "no processing," corresponding to a map $\mathtt{T} \mapsto \varphi := [\![\tau_{jm} > 0]\!]$. The formula object doesn't belong to $\mathcal{P}$, but rather an extended schema $\mathcal{P} + \varphi$. However, the extension induces an equivalence at the level of semantics: $\widehat{\mathcal{P}} \simeq \widehat{\mathcal{P} + \varphi}$, so we can usually ignore the distinction. In particular, we can allow the broader class of maps without loss of generality. This formalizes the logician's observation that a definition can always be elaborated away. Definitions will be useful in Section 5, where we use them to recombine data assembled from different sources.

At the level of solutions we have a similar schema:

$$\mathcal{S}' := \left\langle \begin{array}{ccc} \mathtt{T} & \overset{s'}{\rightrightarrows} & \mathbb{R}^+ \\ {\scriptstyle j}\downarrow & \overset{t'}{\searrow} & \\ \mathtt{J} & \underset{m}{\to} & \mathtt{M} \end{array} \;\middle|\; \text{axioms} \right\rangle.$$

[3] *Technically, $\mathbb{R}^+$-valued terms definable in the schema $\mathcal{S}$. These might be linear combinations, polynomials or more sophisticated expressions depending on which operations are included in* **Type**.

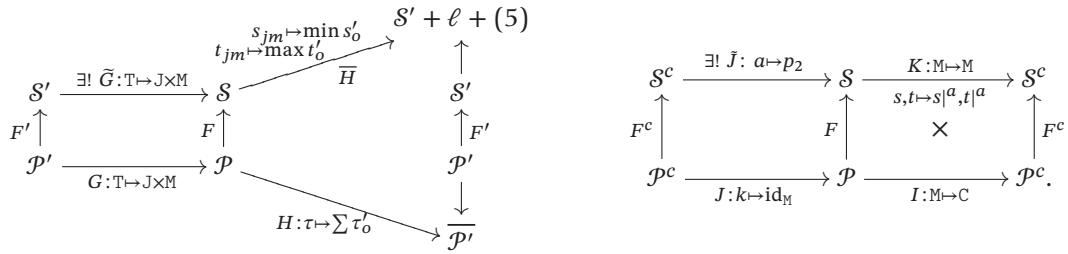


Figure 2. Variants of the open-shop problem from Section 4.1 (tasks) and Section 4.2 (machines), respectively.

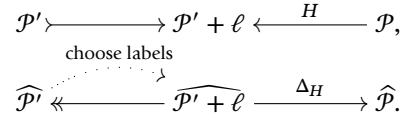
The new no-overlap axioms are quantified over tasks $o \in \mathtt{T}$, while the old contexts (j, j', m vs. j, m, m') are replaced by equational antecedents

$$\begin{aligned} &\text{(a)} \quad \forall o \neq o' \quad \big(m_o = m_{o'} \quad \Rightarrow (t'_o \leq s'_{o'}) \vee (t'_{o'} \leq s'_o)\big), \\ &\text{(b)} \quad \forall o \neq o' \quad \big(j_o = j_{o'} \quad \Rightarrow (t'_o \leq s'_{o'}) \vee (t'_{o'} \leq s'_o)\big). \end{aligned} \tag{1'}$$

There is a unique lift of G to the solution schemas (Notation: $\exists!\ \widetilde{G}$). To fully justify this claim, one must prove, e.g., that original no-overlap condition (1a) entails the new condition (1a′). This uses the fact that $\mathtt{J} \times \mathtt{J} \times \mathtt{M}$, the context object for (1a), is isomorphic to the formula object of the antecedent $[\![m_o = m_{o'}]\!]$ in (1a′).

Things don't go as smoothly in the other direction: there is *almost* a functor $H : \mathcal{P} \to \mathcal{P}'$. One intuitive way to define τ from τ' is to think of $\mathtt{T}$ as a bundle over $\mathtt{J} \times \mathtt{M}$ and sum τ' over the fibers. This feels like a definitional extension, but we'll see in Section 6 that this sort of aggregation requires a subtle piece of additional structure: a labeling of the tasks.

The upshot is that there is a *labeled extension* $\mathcal{P}+\ell$ (which is not definitional!) in which the fiber sum *is* defined. This doesn't get us directly from $\widehat{\mathcal{P}'}$ to $\widehat{\mathcal{P}}$, as one of the functors goes the wrong way:



However, the semantic dual of $\mathcal{P} \rightarrowtail \mathcal{P} + \ell$ is surjective on objects, so we can always choose a labeling for a $\mathcal{P}'$-instance (but not functorially!). Then we apply Δ_H to extract the desired $\mathcal{P}$-problem. In general the result may depend on the choice of labels, though in this case it is invariant by commutativity of the sum.

We can give a similar reduction for solutions, with one extra hitch. In this case, the best option for fiberwise reduction is

$$s_{jm} = \min_{o \in T_{jm}} s'_o, \qquad t_{jm} = \max_{o \in T_{jm}} t'_o.$$

However, this may cause problems if there are gaps between tasks in a fiber. For one, the resulting schedule may violate no-overlap, since other tasks may be active during a gap in the imputed interval $[s_{jm}, t_{jm}]$. Even when no-overlap is satisfied, downtime creates an inconsistency between the two interpretations of τ (through $\mathcal{S}$ and $\mathcal{P}'$, respectively). Coherence requires

$$\max_{o \in T_{jm}} t'_o - \min_{o \in T_{jm}} s'_o = \sum_{o \in T_{jm}} (t'_o - s'_o). \tag{5}$$

The best we can do is adjoin this requirement as an axiom (as well as the labeling of $\mathtt{T}$) and interpret $\mathcal{S}$ in the resulting extension $\mathcal{S}' + \ell + (5)$.

To summarize, we can generalize the traditional open-shop model by replacing the product structure $\mathtt{J} \times \mathtt{M}$ with a new set variable $\mathtt{T}$. The two versions are linked by functors from the variants (problem and solution) into the originals. Under semantic duality, these exhibit the traditional model as a special case of the variant one. We can also reduce a flexible problem to a traditional one, at the cost of introducing some additional structure, by aggregating task-indexed times over the fibers of the bundle $\mathtt{T}$. Flexible solutions, on the other hand, only generate traditional solutions under an additional assumption (5). All of this is succinctly encoded in the left-hand diagram of Figure 2.

4.2. **Duplicate machines.** Our next variation considers a more flexible model of the shop floor, allowing different machines to be treated interchangeably. The problem is that a factory may have several drills, but the traditional model has no way of indicating that job j_1 should be done on $\mathtt{drill}_1$ *or* $\mathtt{drill}_2$ for 20 minutes.

To model the flexible shop we introduce an object of *capabilities* $\mathtt{C}$ along with a bundle $k : \mathtt{M} \to \mathtt{C}$, thought of as an indexed family of sets $\mathtt{M}_c$. For example, a factory instance P with one saw, two drills, three lathes and one mill would send k to the function

$$\begin{array}{ccccccccc} P(\mathtt{M}) & = & \{s_1, & \underbrace{d_1, d_2}, & \underbrace{l_1, l_2, l_3}, & & m_1\} \\ \downarrow & & \downarrow & \downarrow & \downarrow & & \downarrow \\ P(\mathtt{C}) & = & \{\mathtt{saw}, & \mathtt{drill}, & \mathtt{lathe}, & & \mathtt{mill}\}. \end{array}$$

In practice even this is too restrictive, as a machine may have multiple capabilities (modeled as a span $\mathtt{M} \leftarrow \bullet \rightarrow \mathtt{C}$), but we will stick with the simpler version here.

The new problem schema $\mathcal{P}^c$ adds in k and replaces $\mathtt{M}$ by $\mathtt{C}$ in the domain of τ, indicating that a task should be

completed on *some* drill or saw, but not which one:

$$\mathcal{P}^c := \left\langle \begin{array}{ccc} \mathtt{M} & \xrightarrow{\;k\;} & \mathtt{C} \\ \mathtt{J}\times\mathtt{C} & \xrightarrow{\;\tau^c\;} & \mathbb{R}^+ \end{array} \right\rangle.$$

For the solution, we retain the start and stop times $s \le t$, but these are indexed by capabilities rather than machines, so we also need an assignment $a : \mathtt{J}\times\mathtt{C} \to \mathtt{M}$. This should commute over $\mathtt{C}$ (i.e., $a.k = p_2$), enforcing the condition $a(j,c) \in \mathtt{M}_c$:

$$\mathcal{S}^c := \left\langle \begin{array}{ccc} \mathtt{M} & \xleftarrow{\;a\;} & \\ {\scriptstyle k}\downarrow & & \mathtt{J}\times\mathtt{C} \;\underset{t^c}{\overset{s^c}{\rightrightarrows}}\; \mathbb{R}^+ \\ \mathtt{C} & \xleftarrow[\;p_2\;]{} & \end{array} \;\middle|\; \text{axioms} \right\rangle.$$

No-overlap equation (1b) remains unchanged except for modifying the variable names $(j, m, m' \mapsto j, c, c')$. On the other hand (1a) should be weakened to allow simultaneous assignments on parallel machines:

$$\begin{aligned} \forall j \neq j'\ \forall c\ (t_{jc} \le s_{j'c}) \qquad & (1a^c) \\ \vee\ (t_{j'c} \le s_{jc}) & \\ \vee\ (a_{jc} \neq a_{j'c}). & \end{aligned}$$

As before, we can exhibit $\mathcal{P}$ as a special case of $\mathcal{P}^c$ by mapping $k \mapsto \mathrm{id}_{\mathtt{M}}$. This amounts to extending $\mathcal{P}^c$ by an axiom asserting that $|\mathtt{M}_c| = 1$ for all c. Moreover, this functor lifts uniquely to the solution schema, since we have no choice but to assign a to the projection p_2:

$$\begin{array}{ccc} \mathcal{S}^c & \overset{\exists!\ \tilde{J}:a\mapsto p_2}{- - - - \to} & \mathcal{S} \\ {\scriptstyle F^c}\uparrow & & \uparrow{\scriptstyle F} \\ \mathcal{P}^c & \xrightarrow[J:k\mapsto \mathrm{id}_{\mathtt{M}}]{} & \mathcal{P} \end{array}$$

In the opposite direction, $I : \mathcal{P} \to \mathcal{P}^c$ is already a subcategory up to renaming ($\mathtt{M} \mapsto \mathtt{C}$). For solutions, though, sending $\mathtt{M}$ to $\mathtt{C}$ would violate no-overlap. The whole point of duplicate machines is to run them at the same time! Nonetheless, we can obviously turn an $\mathcal{S}^c$-instance into a traditional schedule by viewing unassigned pairs jm as "no processing." Formally, $K : \mathcal{S} \to \mathcal{S}^c$ sends $M \mapsto M$ and defines s and t using extension by zero along a.

The main point here is that even though we can interpret both problem and schedule representations in the new context, we cannot interpret the problem formulation because the two interpretations are inconsistent. They treat machines differently,

$$K(F(\mathtt{M})) = K(\mathtt{M}) = \mathtt{M} \neq \mathtt{C} = I(\mathtt{M}) = F^c(I(\mathtt{M})),$$

so the resulting square fails to commute, as indicated by × in Figure 2.

5. Improvisation

Open-shop scheduling involves three main concepts: jobs, machines and tasks. The previous section considered generalizations of the latter two using the slice construction. Now we turn to duplicate jobs, but with a rather different perspective, focused on the use of colimits (pushouts) to merge schemas and aggregate data from different sources.

In practical scheduling contexts there is often substantial repetition in the jobs to be scheduled. For example, a factory might manufacture a small number of products in large volumes, or a hospital might administer the same battery of tests to many patients. These two pieces of information vary on different timescales; the specific orders or patients change day-to-day, but process plans might be updated annually (or never). Consequently, it can be useful to separate the two data sources and reassemble them on-the-fly as needed.

We start with a process catalog $C \in \widehat{\mathcal{P}}$ describing all the jobs available for production. $\mathcal{P}$ is the same category we started with in Section 2, but now $C(\mathtt{J})$ represents a set of offerings rather than specific jobs to be completed.

Information on specific orders is kept in a separate database $O \in \widehat{\mathcal{O}}$ containing customer information:

$$\mathcal{O} := \left\langle \begin{array}{ccccc} \mathtt{C} & \xleftarrow{\;c\;} & \mathtt{O} & \xleftarrow{\;o\;} & \mathtt{I} \\ {\scriptstyle b}\downarrow & \swarrow{\scriptstyle r} & \swarrow{\scriptstyle d} & & \downarrow{\scriptstyle j} \\ \mathbb{R}^+ & & \xleftarrow[\;k\;]{} & & \mathtt{J} \end{array} \right\rangle.$$

Every customer $\mathtt{C}$ has a set of (unfilled) orders $\mathtt{O}$, each with deadline d, and every order includes a set of items $\mathtt{I}$. Each item requires the completion of a particular job j from the process catalog, which is billed at cost k. The other numeric attributes r and b represent rollover balances and billing amounts, formalized (using aggregation, per Section 6) in terms of a constraint,

$$b = r + \sum_{o:O_c} \sum_{i:\mathtt{I}_o} k(j(i)).$$

Of course, much of this information is irrelevant for our purposes, but this is meant to suggest that $O \in \widehat{\mathcal{O}}$ is a legacy data system already in use for other purposes.

Though we refer to j as a job "from the process catalog," officially name-spaces are disjoint; after all, we *don't* want to identify the customer object $\mathtt{C}$ here with the capability object $\mathtt{C}$ from Section 4. Consequently, conceptual overlap must be explicitly represented as an overlap schema together with functors into each of the components. In this case, the overlap is just a one-object category $\{\mathtt{J}\}$, with the obvious inclusions into $\mathcal{P}$ and $\mathcal{O}$.[4]

[4] *We overload notation by writing* $\mathtt{J}$ *for* both *inclusion functors* $\mathcal{P} \leftarrow \{\mathtt{J}\} \to \mathcal{O}$. *In particular, both pairs of change-of-base adjunctions are written* $\Sigma_{\mathtt{J}} \dashv \Delta_{\mathtt{J}}$, *but these can always be disambiguated based on context.*

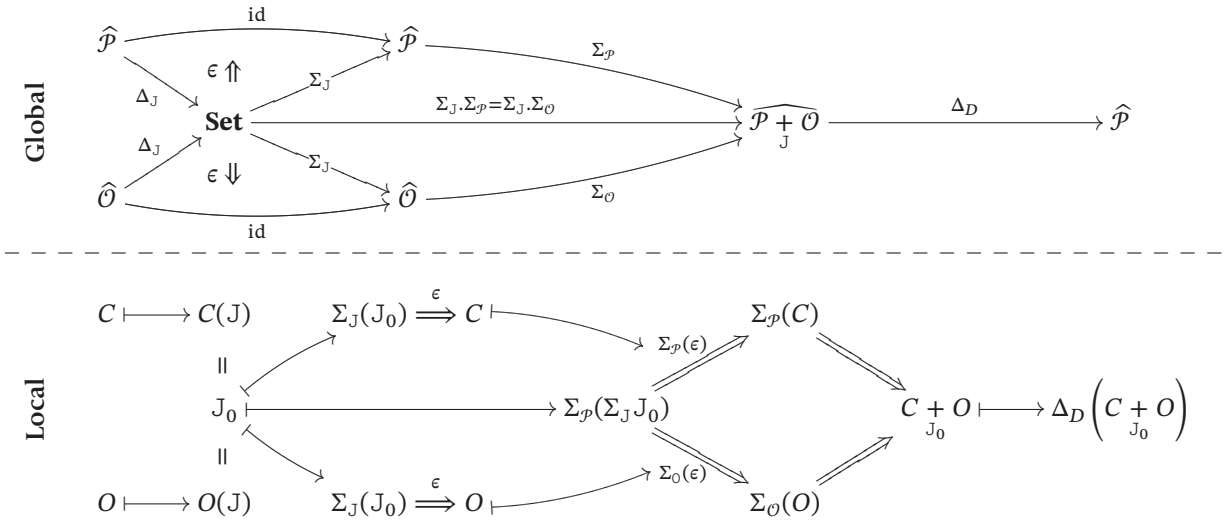


Figure 3. Local and global views of the categorical data integration workflow.

Using the overlap we can form a *pushout* schema $\mathcal{P} +_{\mathtt{J}} \mathcal{O}$. The pushout is a categorical construction that generalizes set-theoretic union [Spi14]. It contains all the objects, arrows, equations and structure in $\mathcal{P}$ and $\mathcal{O}$, with an equation for each element of the overlap (i.e., $\mathtt{J}_{\mathcal{P}} = \mathtt{J}_{\mathcal{O}}$). However, any name-space collisions that are not in the overlap must be disambiguated.

Note that we are not assuming the presence of pushouts within our schemas (although we could). Instead, we are using pushouts as an algebraic construction in the category of schemas and functors. One can prove that schemas are closed under pushouts (and other colimits) because structured categories and functors are themselves equationally defined (essentially algebraic structures) [AR94].

Next we construct a second functor[5] $D : \mathcal{P} \to \mathcal{P} +_{\mathtt{J}} \mathcal{O}$, distinct from the pushout inclusion, which extracts a daily schedule by combining the two data sets. The items $\mathtt{I} \in \mathcal{O}$ define the day's jobs ($\mathtt{J} \mapsto \mathtt{I}$), while the processing times are calculated by composing j with τ:

$$D(\mathtt{J} \times \mathtt{M}) \cong \mathtt{I} \times \mathtt{M} \xrightarrow[j \times \mathrm{id}]{} \mathtt{J} \times \mathtt{M} \xrightarrow{\tau} \mathbb{R}^+, \qquad D(\tau) : D(\mathtt{J} \times \mathtt{M}) \to \mathbb{R}^+.$$

These schemas and functors establish a static set of relationships that can be used to operate dynamically on data using the base-change operations (Δ, Σ) from Section 3.2. The workflow to integrate C and O is traced out in Figure 3, and a more detailed description of the integration methodology can be found in [BPS19].

The first step is to establish a semantic overlap between instances, as a complement to the conceptual overlap established for schemas. Δ-operations (inverse image) pull C and O back to the overlap schema, where they can be compared. Since $\{\mathtt{J}\}$ is a one-object category its instances are just sets, and the Δ-operations are evaluation at $\mathtt{J}$. For simplicity we will assume that C and O have exactly the same set of jobs, $C(\mathtt{J}) = O(\mathtt{J})$, but the procedure we describe works just as well for partial matchings.

Next we use Σ-operations (direct image) to move the semantic overlap back to the component schemas, where it can be compared with the original instances (via the *co-unit* transformation $\epsilon : \Sigma \circ \Delta \Rightarrow \mathrm{id}$). Applying a second round of Σ-operations moves everything to the pushout schema, where we obtain an instance-level pushout over the schema-level pushout $\mathcal{P} +_{\mathtt{J}} \mathcal{O}$.

We abuse notation by writing $C +_{\mathtt{J}_0} O$ for the instance-level pushout, suppressing base-change in the notation. The pushout gives a universal solution to the data integration problem defined by C, O and their overlap, in the sense that any dataset S that extends C and O and agrees on the restrictions to $\{\mathtt{J}\}$ receives a unique map $C +_{\mathtt{J}_0} O \Rightarrow S$. Finally, we extract the daily schedule by applying Δ_D to this merged instance.

Though it does not arise here (by construction), colimit-based integrations often introduce labeled nulls. These can arise from the Σ-operations, as in Section 3, or from the instance-level pushout. Ultimately this should be no surprise, as we expect that merged data sets will be missing information any time they associate different attributes to the same conceptual entities.

To summarize, CT offers a detailed and rigorous procedure for integrating arbitrary datasets. The process depends on an external specification of semantic overlap, which is a hard problem in general, but often easily solved by hand or heuristic in practice. Once the overlap is specified, colimits and change-of-base functors define a data

[5] *More precisely, the functor maps into a (Morita-equivalent) definitional extension of* $\widetilde{\mathcal{P} +_{\mathtt{J}} \mathcal{O}}$*, as discussed in Section 4.1.*

transformation pipeline computing each day's schedule from two loosely-coupled data sets, the process catalog $C \in \widehat{\mathcal{P}}$ and the orders database $O \in \widehat{\mathbb{0}}$.

6. Optimization

6.1. **Objectives.** Everything up to this point has considered scheduling as a constraint satisfaction problem, but in this respect the problem is rather trivial: we can satisfy no-overlap by doing the tasks one at a time. Of course this is clearly unsatisfactory, as it fails to optimize.

For optimization, we first require an objective function. In scheduling, the typical objective is to minimize the *makespan*

$$C_{\max} := \max_{jm}\{t_{jm}\} - \min_{jm}\{s_{jm}\}.$$

The schedule given in Figure 1 has a makespan of 10 hours, which is clearly optimal given that job j_2 alone takes that long.

Other common choices include the average and worst-case *flow times* and the *idle time*:

$$\begin{aligned} f^a &= \frac{1}{|J|}\sum_j \{\max_m\{t_{jm}\} - \min_m\{s_{jm}\}\}, \\ f^{\max} &= \max_j \{\max_m\{t_{jm}\} - \min_m\{s_{jm}\}\}, \\ i &= \sum_m \Big[\max_j\{t_{jm}\} - \min_j\{s_{jm}\} \\ &\qquad - \sum_j (t_{jm} - s_{jm}) \Big]. \end{aligned}$$

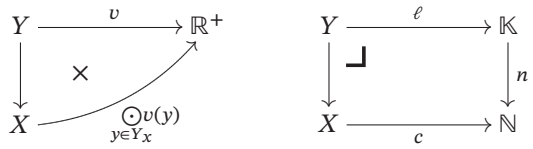
These are particularly relevant in situations like continuous production, where the makespan might not make sense. More generally, we can aggregate flow times using any ℓ^p-average for $p \geq 1$, which tunes the sensitivity to delays.

Once we have an initial stock of objective functions, we can build others by applying sums, linear combinations and other operators. For example, *by themselves* flow times and idle times make terrible objectives: the first can be optimized by doing jobs one at a time, the second by finishing all operations on one machine before moving on to the next. However, the sum $f^a + i$ balances the need to finish jobs quickly but also operate in parallel.

In many cases the objective function will assume some additional structure in $\mathcal{P}$, like a *deadline* $d : \mathtt{J} \to \mathbb{R}^+$. Intuitively, we ought to be able to handle deadlines by adding an additional axiom to the solution schema: $t_{jm} \leq d_j$. This would correspond to a *hard deadline*, with no credit for late delivery.

Sometimes this is too restrictive and we would prefer to build *soft deadlines* into the objective function. Define the *tardiness* of a job j by

$$l_j = \max_j\{0,\ \max_m(t_{jm}) - d_j\}.$$



(a) (b)

Figure 4. (a) Aggregating a value v over the fibers Y_x. (b) The labeling ℓ expresses the bundle Y_x as a pullback.

The *tardiness penalty* is then $\sum_j h(l_j)$, where h is one of several penalty functions:

$$h(x) = \begin{cases} \chi_{x>0} & \text{(one-time penalty)}, \\ \max\{x, 0\} & \text{(linear penalty)}, \\ \max\{x^2, 0\} & \text{(quadratic penalty)}, \\ \max\{x, -1\} & \text{(early completion bonus)}. \end{cases}$$

We can also vary the aggregation strategy, replacing $\sum_j$ by $\max_j\{\ \}$ or other ℓ^p combinations.

Formally, we would like to represent the objective function as an arrow $1 \xrightarrow{f} \mathbb{R}^+ \in \mathcal{S}$. Although both objects belong to the subcategory **Type**,[6] f typically depends on structure specific to $\mathcal{S}$. When we interpret f in an instance $S \in \widehat{\mathcal{S}}$, this picks out a single element $S(f) \in S(\mathbb{R}^+)$. If S is a true solution, i.e., a ground instance with no missing data, then $S(\mathbb{R}^+) = \mathbb{R}^+$ and $S(f) \in \mathbb{R}^+$ is the objective value of S. Otherwise S is a template, $S(\mathbb{R}^+)$ is a set of real-valued functions, and $S(f)$ is an explicit representation of the objective function in terms of the free variables in S.

There is just one problem with this approach: aggregations like $\sum_j$ and $\max_m$ are not definable in $\mathcal{S}$.

6.2. **Aggregation.** At several points now we have encountered the same situation: we have a bundle $Y \to X$ and some attribute value $v : Y \to \mathbb{R}^+$ that we would like to aggregate using sums, maxima or some other combination $\odot$, in order to define an auxiliary attribute on X, as shown in Figure 4(a). However, first-order logic is not sufficient to define these aggregates, which require more powerful machinery from the categorical theory of computation and higher-order logic. Our goal is to mimic the "folding" operation used to aggregate lists, trees and other "containers" in functional programming.

The main element in our construction is the ordering relation on $\mathbb{N}$, viewed as a bundle

$$\mathbb{K} = [\![k < n]\!] \xrightarrow{n} \mathbb{N}.$$

$\mathbb{K}$ is called a *universe* or small-object classifier because any finite bundle can be expressed as a pullback of $\mathbb{K}$,[7] as

[6] 1 *is the* terminal object, *the categorical analogue of a singleton in set theory.*

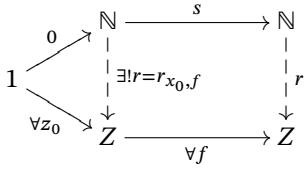
[7] *In* **Set**, *the pullback of a bundle* $Z \to X$ *along an arrow* $f : Y \to X$ *is the bundle over* Y *defined fiberwise by* $(f^*Z)_y = Z_{f(y)}$. *Similarly, a family of*

shown in Figure 4(b). We refer to this pattern as a *labeling*, and we usually refer to the entire pattern as ℓ, leaving c (and the pullback condition) implicit.

This is reasonable because c is uniquely determined. Pullbacks copy fibers and $|\mathbb{K}_n| = n$, so the pullback condition can only be satisfied if $c : x \mapsto |Y_x|$. On the other hand, there are many choices for the specific labeling ℓ. Each choice corresponds to a family of linear orders on the fibers Y_x, so there are $\prod_x |Y_x|!$ in total.

There is a subtle point: although every finite bundle has a unique cardinality c, adding c to a schema is *not* a definitional extension. It changes the model's semantics, restricting the meaning of a transformation between instances. Roughly, mappings without a labeling are commutative squares; mappings that preserve labels must be pullbacks.

Given a labeling ℓ we can construct the desired aggregations using the recursive structure of $\mathbb{N}$. A recursive function $r : \mathbb{N} \to Z$ is typically defined by a base case $z_0 \in Z$ and a recursive formula $r(n+1) = f(r(n))$. Diagrammatically:

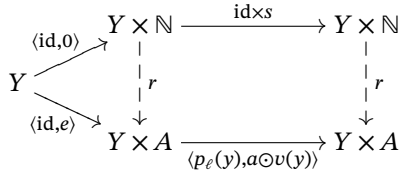


A slightly more complicated variant allowing parameterized recursion is sufficient to define all total recursive functions on $\mathbb{N}$ [Mai10].

Theorem 1. *Fix a monoid $(A, \odot, e)$. The aggregate of an attribute $v : Y \to A$ over a bundle $Y \to X$ is definable using an $\mathbb{N}$-labeling. If $\odot$ is commutative, the result does not depend on the choice of ℓ.*

Proof. Write p for the fiberwise predecessor on $\mathbb{K}$, sending $\langle k, n\rangle \mapsto \langle \max\{0, k-1\}, n\rangle$. Viewing this as an n-indexed family of functions and pulling back along the labeling induces a map $p_\ell : Y \to Y$ that traverses the fibers of Y.

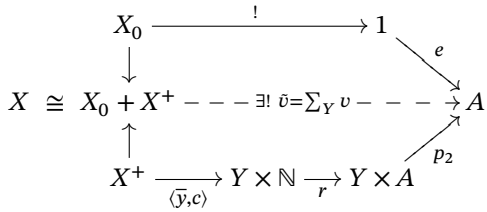
Next we construct a recursive function r, parameterized by Y, that maintains a running aggregate of v-values. The target of the function is $Y \times A$, and the recursive step (bottom map below) advances y along p_ℓ and combines the previous aggregate a with the next v-value: $\langle y, a\rangle \mapsto \big\langle p_\ell(y), a \odot v(y)\big\rangle$.



functions $h_x : Z_x \to Z'_x$ pulls back to $(f^*h)_y = h_{f(y)}$, so pullback is functorial. In diagrams, the pullback is often indicated by a corner notation ($\lrcorner$), as in Figure 4(b).

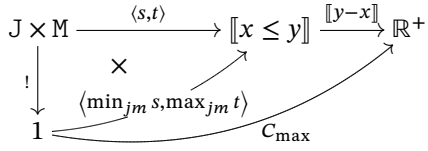
Unrolling the definition, we see that the first component iterates p_ℓ, while the second computes $e \odot v(y) \odot v(p_\ell(y)) \odot \ldots \odot v(p_\ell^{n-1}(y))$. Evaluating at the correct input will realize our desired aggregation. Specifically, the "top" element $\overline{y}(x) \coloneqq |Y_x| - 1$ in each fiber is defined via the equivalence $Y_x \cong (c^*\mathbb{K})_x \cong \mathbb{K}_{c(x)} \cong [\![k < c(x)]\!]$.

The only remaining issue is a case split to handle empty fibers. This corresponds to a coproduct decomposition[8] $X = X_0 + X^+$, where $X_0 = [\![c(x) = 0]\!]$ and $X^+ = [\![c(x) > 0]\!]$. The top-of-fiber map $\overline{y}$ is only defined on X^+, but everything else can be mapped to the identity $e \in A$:



In general, the result of an aggregation will depend on the choice of labeling ℓ, as this defines the order in which $\odot$ is applied. If $\odot$ is commutative the order is irrelevant, and the aggregate is independent of ℓ. Formally, this is proved by induction on the fiber cardinality. □

Now we can use these aggregates to define our objective functions within some labeled extensions of the solution category $\mathcal{S}$. For example, the makespan is defined by folding $\min \times \max$ over $\mathtt{J} \times \mathtt{M}$ and taking the difference of the results:



7. Solving the Problem

We conclude with a formulation of problem solving in categorical terms. We focus on the classical open-shop problem $F : \mathcal{P} \to \mathcal{S}$ from Section 2, but the approach is broadly applicable.

Scheduling problems are almost universally NP-hard, except in some very restrictive special cases (e.g., only two machines), so practical solutions must rely on heuristics and approximations. We present one of the simplest, a matching algorithm adapted from [BHM+08], where many others can also be found. Without loss of generality, we will assume that $|\mathtt{J}| \le |\mathtt{M}|$; otherwise, the roles of the two objects should be reversed.

[8] *The* coproduct $X + Y$ *is a categorical constructor that corresponds to* disjoint union *in set theory. Each summand has an* inclusion *arrow $X \to X + Y \leftarrow Y$, and the universal property of the coproduct defines a* copairing *constructor $[f, g]$ that formalizes definition by cases. The n-ary coproduct is denoted $\coprod_i X_i$.*

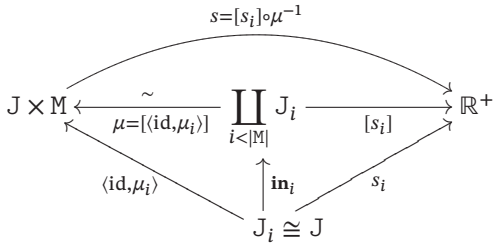


Figure 5. The matching algorithm in Section 7 constructs a factorization of s through an $|M|$-fold coproduct of $\mathtt{J}$.

First the algorithm defines a rank order on the tasks in each job:

$$\overbrace{\mu_0(j) \rightsquigarrow \mu_1(j) \rightsquigarrow \ldots \rightsquigarrow \mu_{|\mathtt{M}|-1}(j)}^{\text{Machine sequence for job } j}$$

Here μ_i, called a *matching*, is just a monic $\mathtt{J} \rightarrowtail \mathtt{M}$. It assigns a machine to each job, with no overlap. These can be ranked according to a number of heuristics; the simplest just minimizes $\sum_j \tau_{j,\mu(j)}$, based on the intuition that tasks of similar size should be grouped to minimize idle time. An explicit tie-breaking mechanism is also required.

Using this, we generate a disjoint sequence of matchings. Two matchings μ and μ' *meet* if $\mu(j) = \mu'(j)$ for some $j \in \mathtt{J}$, i.e., if their graphs intersect in $\mathtt{J}\times\mathtt{M}$. Write W_0 for the set of all matchings $\mathtt{J} \rightarrowtail \mathtt{M}$ and select the minimal element μ_0 according to the chosen heuristic. Now remove any matchings that meet μ_0 to obtain $W_1 \subseteq W_0$, and choose the minimal element $\mu_1 \in W_1$. Repeat to define $\langle \mu_i \rangle_{i<|\mathtt{M}|}$.

Next, we assign start times $s_i : \mathtt{J} \to \mathbb{R}^+$ to each matching μ_i, starting from $s_0 \equiv 0$. As usual, stop times are calculated from τ: $t_i := s_i + \tau(\mathrm{id}, \mu_i)$.

To ensure no-overlap, we also keep track of release times for the machines, given by

$$r_i(m) = \begin{cases} t_i(\mu_i^{-1}(m)) & \text{if } m \in \mu_i(J), \\ r_{i-1}(m) & \text{o.w. (base case 0).} \end{cases}$$

Putting these together, we obtain the earliest starting time for the next stage: $s_i = \max\{t_{i-1},\ r_{i-1} \circ \mu_i\}$.

Categorically, this definition describes a factorization of s through a coproduct decomposition, as shown in Figure 5. The rank orders define an isomorphism $\mu := [\langle \mathrm{id}, \mu_i \rangle]_{i<|M|}$ (by disjointness and cardinality considerations), and its inverse composes with the start times to give $s = [s_i] \circ \mu^{-1}$.

We can (almost) interpret this algorithm as a functor $A : \mathcal{S} \longrightarrow \mathcal{P}$. It sends elements of the solution category (i.e., s, t) to constructions in the problem category (built from τ). It preserves structure (because the no-overlap axioms are satisfied).

This formulation is semantically sensible. Δ_F displays schedules as a bundle over the problems, and the algorithm provides a mapping back in the other direction. Moreover, we can think of the solution as an indexed element of the fiber $A_P \in \widehat{\mathcal{S}}_P$. In CT, such a map is called a *section* of the bundle, defined by the equation $\Delta_F \circ \Delta_A = \mathrm{id}$. This follows immediately from the dual condition on syntax,

$$A(F(\tau)) = A(t - s) = (s + \tau) - s = \tau.$$

The only wrinkle is that Δ_A is not functorial; the solution of a subproblem is not, in general, a subschedule of the original solution. Formally, this reflects the fact our construction relies extensively on labeling; the target of A is a labeled extension $\mathcal{P} + \ell$ rather than $\mathcal{P}$ itself.

Semantically, $\widehat{\mathcal{P} + \ell}$ defines a fibration over the *core* of $\widehat{\mathcal{P}}$, the subcategory of objects and *iso*morphisms. All maps are bijections because they must preserve labels, and hence cardinality. Fibrations are defined by lifting properties, which in this case follow from the fact that we can transport labels across isomorphisms. Consequently, we can use A to lift problems (and isomorphisms) of $\widehat{\mathcal{P}}$ to schedules in $\widehat{\mathcal{S}}$, though the result will depend on the choice of labeling on the problem instance.

Conclusion

We now have an extensive model of the open shop:

§2 Problem representation $\mathcal{P}$
§3 Problem formulation $F : \mathcal{P} \to \mathcal{S}$
§4 Problem variation $\mathcal{P} \underset{J}{\overset{I}{\rightleftarrows}} \mathcal{P}^c$
§5 Problem extraction $\Delta_D(C +_{\mathtt{J}_0} O)$
§6 Solution optimization $C_{\max} : 1 \to \mathbb{R}^+$
§7 Solution algorithm $A : \mathcal{S} \to (\mathcal{P} + \ell)$

To develop our model, we introduced categories, functors, constuctors, duality, change of base, bundles, meta-constructors, universes and aggregation. This is quite a lot of theory and, though most is elementary, it may be a bit overwhelming all at once. However, the intricacy is necessary to support many of CT's unique features as a modeling language:

- Built-in computation (**Type**)
- Built-in logical constraints (structures)
- Unified syntax/semantics (duality)
- Schema mappings (functors)
- Meta-theory (schema diagrams, constructors)
- Data migration (change of base)
- Data integration (colimits/pushouts)

The payoff from all this abstraction is flexibility. In an industrial scheduling problem, the specific requirements range from details of the physical process (e.g., color contamination in vinyl flooring manufacture) to customer-specific process restrictions (e.g., disabled-worker subsidy for government orders). Moreover, these requirements change on a regular basis.

Textbook solutions rarely suffice and, when they do, non-trivial translation is still needed to place the problem into textbook form. In practice, what we would like is a rigorous method to specify and refine the process requirements *in context* and link these to existing databases and solvers. What we have outlined here are some first steps in this direction. Hopefully it is clear that the methodology we advance is not limited to scheduling; CT provides a promising language for information modeling and management much more generally.

From here there are many directions for future work. Within scheduling, one could model other "old standards" like job-shop and flow-shop scheduling, and examine their relationships between them. Of particular interest is the degree to which variations (like allowing duplicate machines) translate between different models. Other categorical representations, notably string diagrams, ought to provide a useful view on more complicated issues like multi-resource scheduling (e.g., jobs and workers).

More broadly, the meta-theory that we developed here could provide a framework to formalize a very general class of problems in operations research. Testing the approach against some other classic problems from outside scheduling would be a first step, aiming to identify any gaps in the proposed approach.

However, the most important direction for future work is implementation. The ability to define, examine and manipulate these structures *in silica* is critical for both real-world impact and conceptual accessibility. Computerized interactions can enforce the intricacies of the categorical data model, leaving the user free to explore the space of models. For us, this paper represents a detailed use case to guide such an implementation, demonstrating both the affordances of categorical modeling and the underlying machinery that makes it work.

Disclaimer: Commercial products are identified in this article to adequately specify the material. This does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply the materials identified are necessarily the best available for the purpose.

References

[AR94] Jiří Adámek and Jiří Rosický, *Locally presentable and accessible categories*, London Mathematical Society Lecture Note Series, vol. 189, Cambridge University Press, Cambridge, 1994. MR1294136

[BHM+08] Heidemarie Bräsel, André Herms, Marc Mörig, Thomas Tautenhahn, Jan Tusch, and Frank Werner, *Heuristic constructive algorithms for open shop scheduling to minimize mean flow time*, European J. Oper. Res. **189** (2008), no. 3, 856–870, DOI 10.1016/j.ejor.2007.02.057. MR2400912

[BJS19] Spencer Breiner, Albert Jones, and Eswaran Subrahmanian, *Categorical models for process planning*, Computers in Industry **112** (2019), 103–124.

[BMRPS20] Spencer Breiner, Olivier Marie-Rose, Blake Pollard, and Eswaran Subrahmanian, *Modeling hierarchical systems with operads*, In John Baez and Bob Coecke, editors, Applied Category Theory 2019. Electronic Proceedings in Theoretical Computer Science, 2020. In press.

[BPS19] Spencer Breiner, Blake Pollard, and Eswaran Subrahmanian, *Functorial model management*, International Conference on Engineering Design (ICED 2019), 2019, pp. 1963–1972.

[BPS20] Spencer Breiner, Blake Pollard, and Eswaran Subrahmanian, *Workshop on applied category theory: Bridging theory and practice*, Special Publication 1249, National Institute of Standards and Technology, February 2020.

[FL19] Henrik Forssell and Peter Lefanu Lumsdaine, *Constructive reflectivity principles for regular theories*, J. Symb. Log. **84** (2019), no. 4, 1348–1367, DOI 10.1017/jsl.2019.70. MR4045979

[Mai10] Maria Emilia Maietti, *Joyal's arithmetic universe as list-arithmetic pretopos*, Theory Appl. Categ. **24** (2010), No. 3, 39–83. MR2610177

[Mak88] Michael Makkai, *Strong conceptual completeness for first-order logic*, Ann. Pure Appl. Logic **40** (1988), no. 2, 167–215, DOI 10.1016/0168-0072(88)90019-X. MR972521

[SSVW17] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky, *Algebraic databases*, Theory Appl. Categ. **32** (2017), Paper No. 16, 547–619. MR3641249

[Spi14] David I. Spivak, *Category theory for the sciences*, MIT Press, Cambridge, MA, 2014. MR3288752

Spencer Breiner

Peter Denno

Eswaran Subrahmanian

Credits

All figures are courtesy of the authors.
Photo of Spencer Breiner is courtesy of Spencer Breiner.
Photo of Peter Denno is courtesy of Grace Denno.
Photo of Eswaran Subrahmanian is courtesy of Eswaran Subrahmanian.