USER GUIDE

SCOTT MORRISON AND DAVID SPIVAK

1. TODO

- Write this TODO list.
- Write some 'recipes'; essentially whatever is currently the most impressive demo.

1.1. **Metaphor.**

- The Metaphor section is getting ahead of reality; add trello TODO items.

1.2. **Scala.**

- Have the `\type{...}` command actually link directly to the Scaladoc.
- Explain about `objectsAtLevel`.
- Draw more of the `Functor` type hierarchy.

2. Recipes

3. Metaphor

The Metaphor server (running at `http://categoricaldata.net/metaphor`) provides an interface to the underlying Scala library. It works both as a user interface accessible through a web browser and as a RESTful web service, transmitting objects as JSON data.

The primary objects in Metaphor are *ontologies*, *datasets* and *translations*. Nearly all of the URLs provided by the Metaphor server correspond to one of these three types of objects. Every ontology, dataset or ontology has a JSON representation (a quasi-human readable, machine parseable format, c.f. §3.4). If a client requests such a URL, with the header `Accept: application/json`, the server will respond with the JSON representation of the corresponding object. If you load such a URL in a web browser (which will typically provide a header like `Accept: text/html`) then a human readable web page displaying the object will be returned instead. (Note that this means you can always copy and paste a Metaphor URL from your browser into any input field, or pass it as an argument just as if it were pure JSON.)

3.1. **Examples.**

Currently the Metaphor server provides a small supply of example data. See

- `examples/ontologies`
- `examples/datasets`
- `examples/translations`

for lists of examples, or go directly to `examples/translations/Skip/5/3`.

3.2. **Computations.** The Metaphor server currently supports computation of the three basic data functors (pullback, left and right pushforward), via requests to

- `compute/pullback`
- `compute/leftPushforward`
- `compute/rightPushforward`

Each of these requests requires two parameters, `translation` and `dataset`. The value of these parameters may either be a JSON encoded object or an encoded URL. If it is a URL, the Metaphor server will make the necessary requests to obtain the desired object. (In fact, it might take some shortcuts, if it's a URL pointing to the same Metaphor server.)

Requesting these URLs with missing or invalid parameters from a browser will produce a form asking for further input.

3.3. **External URLs.** It's perfectly possible to store ontologies, translations and datasets outside of the Metaphor server. Any URL that resolves to a JSON encoded object will do.

3.4. **JSON formats.**

4. Scala

4.1. **Class hierarchies.** This section contains an overview of the important types in the Scala library. You should read this section in conjunction with the Scaladocs.

The three most important types are

- `net.metaphor.api.Ontology`
  All database schemas have type `Ontology`.
- `net.metaphor.api.Translation`
  A `Translation` is a functor between two `Ontology`s
- `net.metaphor.api.Ontology#Dataset`
  A `Dataset` is a functor from an `Ontology` to `Set`. (Recall in Scala the # denotes an inner class — thus every `Dataset` is attached to a particular `Ontology` instance.)

`Ontology` is a subtype of a long sequence of more general classes of categories, illustrated in Figure 1.

4.2. **Categories.** At the top of the hierarchy of categories is `Category`. It contains two abstract type members, `O` and `M`, which represent the types of objects and morphisms for the category. (If a type has an abstract type member, it cannot be instantiated — some subtype will override these type members, specifying concrete types.) Essentially the only other functionality in `Category` are the methods `identity`, `source`, `target` and `compose`, which provide the basic operations on objects and morphisms.

Below `Category` we have `SmallCategory`. For our purposes, a `SmallCategory` is a category for which we can talk about the category of functors to `Set`. In particular, `SmallCategory` contains an inner type `FunctorToSet`. (This will eventually be specialized to the inner type `Dataset` in `Ontology`.)

Below `SmallCategory` we have `LocallyFinitelyGeneratedCategory`. Mathematically, a locally finitely generated category is a category with a finite set of 'generators' between each pair of objects, such that every morphism can be obtained by composing some sequence of generators. Moreover, we insist that the set

of generators with a fixed source but arbitrary target is finite (even when there are infinitely many objects), and similarly for a fixed target.

In Scala, we implement this by introducing a new abstract type member `G` to represent generators, and override the type `M`, defining it once and for all to be `PathEquivalenceClass`. Further, `LocallyFinitelyGeneratedCategory` provides definitions of `identity`, `source`, `target` and `compose`, the basic operations from `Category`. An implementation of a `LocallyFinitelyGeneratedCategory` must provide a number of new methods, instead. The most important of these is `def generators(s: O, t: O): List[G]`, specifying the generators between two objects. Further, the category must provide a method `def pathEquality(p1: Path, p2: Path): Boolean`, which determines whether two compositions of generators are equal. (Override the method `def pathHashCode(p: Path): Int` is highly recommended as well.)

Below `LocallyFinitelyGeneratedCategory` we have `FinitelyGeneratedCategory`. Now we insist that there are finitely many objects.

Below `FinitelyGeneratedCategory` we have `FinitelyPresentedCategory`, which provides a method `def relations(s: O, t: O): List[(Path, Path)]`. In principle at least, `FinitelyPresentedCategory` could provide an implementation of the method `pathEquality`, but since this is potentially a hard (!) problem we defer this to traits which use particular strategies to decide path equality.

Finally, an `Ontology` inherits from `FinitelyPresentedCategory`.

4.3. **Functors.** The basic `Functor` type is

```
trait Functor {
    val source: Category
    val target: Category

    def onObjects(source.O): target.O
    def onMorphisms(source.M): target.M
}
```

It also comes with some convenience `apply` methods, so if `F` is a functor we can simply write `F(o)` or `F(m)` to apply it to an object or morphism. Implementations, however, should override `onObjects` and `onMorphisms`.

The type hierarchy is quite complicated, and two-dimensional rather than linear! Most of the relevant types are traits contained in the object `Functor`. In particular, we have

- `Functor.withSmallSource`
- `Functor.withLocallyFinitelyGeneratedSource`
- `Functor.withFinitelyGeneratedSource`
- `Functor.withFinitelyPresentedSource`

(each inheriting from the last) and

- `Functor.withSmallTarget`
- `Functor.withLocallyFinitelyGeneratedTarget`
- `Functor.withFinitelyGeneratedTarget`
- `Functor.withFinitelyPresentedTarget`

along with the combined types `Functor.withXXXSource.withYYYTarget`, for each of the possible values of `XXX` and `YYY`. This two-dimensional hierarchy, along with the most important features of some types, is shown in Figure 2.

`Functor.withLocallyFinitelyGeneratedSource` (and hence all its descendants) provides an implementation of `onMorphisms`, but has a new abstract method `def onGenerators(source.G): target.M` which must be implemented by instances.

See below in §4.5 for a description of where in this type hierarchy the various data functors (pullback, left and right pushforwards) are defined.

4.4. **The category of `Sets`.**

4.5. **Functors to `Set`.**

4.6. **Data functors.**

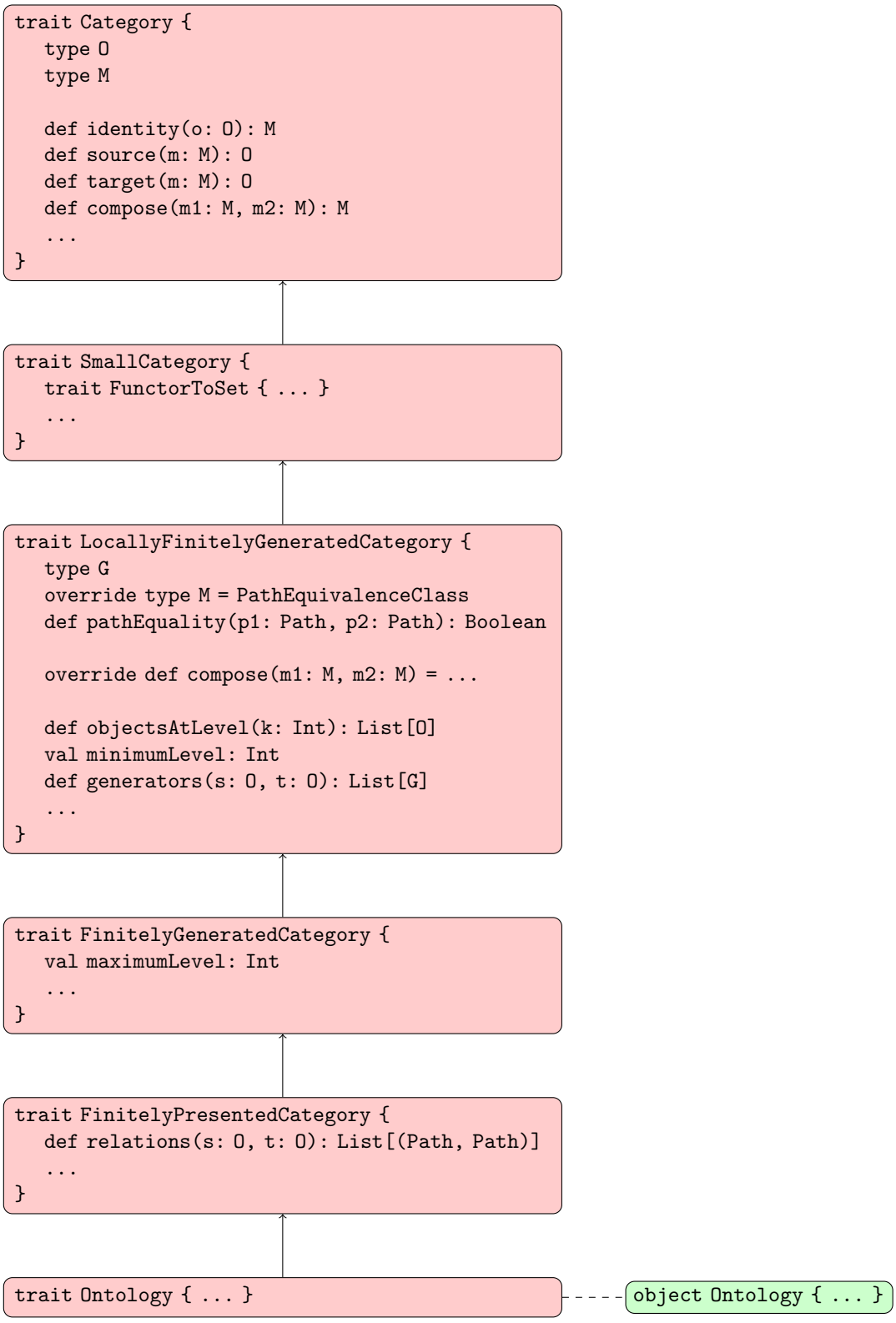

FIGURE 1. The `Ontology` type hierarchy.

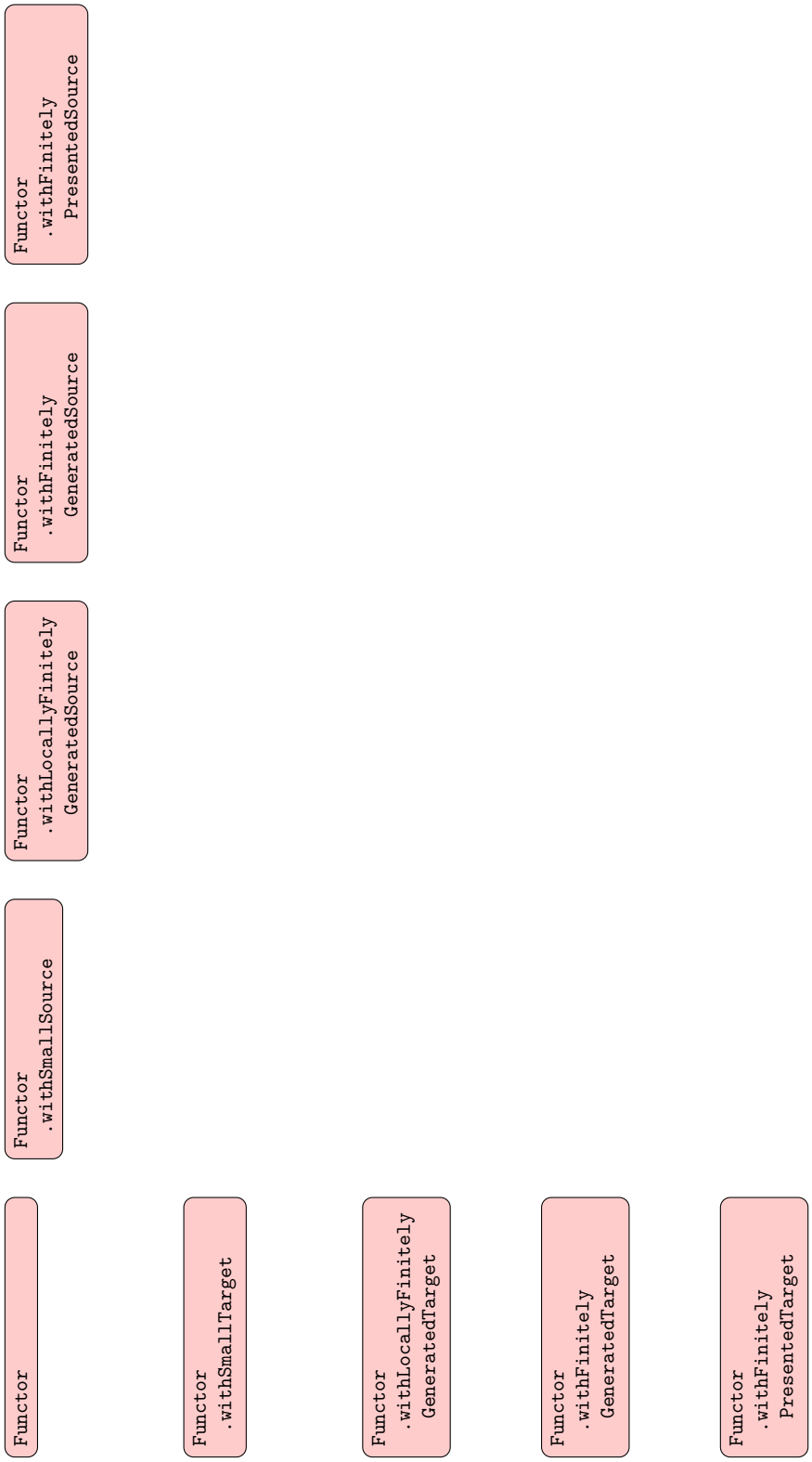

FIGURE 2. The `Functor` type hierarchy.