Primitives:

1) Data type
2) Set (mathematical)
3) Element (mathematical)

Axioms:

1) For each $type_i$, a function $type_i(x)$ returns true iff x is of type $type_i$.

Progression of Components:

1) type $\in TYPES = \{integer, boolean, string, object^*\}$
2) Datum = (type, value)
3) Method (q): $f(type^*) \rightarrow datum^*$
4) Property (y): $(name, type) \mid name \in String$
   a) Property instance (v_i):

   $([\![name]\!]_i, [\![type]\!]_i, [\![value]\!]_i) \mid [\![type]\!]_i ([\![value]\!]_i) = true$
5) Object (B) : $(name, \{p^*\}, \{m^*\})$
   a) Object instance (B_i): $(id, B, \{p_i | j^*\})$
6) Object Method (m_B): $f(B, type^*) \rightarrow (B', datum^*)$
   a) Object Method instance $: m_B(B_i) \rightarrow (B_i')$
7) Key (k) : $k = \{p | \forall p_i, p_j : value_i = value_j \Rightarrow i = j\}$
8) Keyed Object (o): $B \; s.t. \exists p \in B | p \; is \; a \; k$
   a) Keyed object instance (o_i): $(id, v, \{k_i, p_{ij}^*\})$
9) Application $(Appl)$: $(name, \{B^\dagger *\}, \{(B \times B)^\dagger *\}, \{y^\dagger *\})$
10) Container

Comment [s1]: I'm still not convinced this is really necessary to our algebra… I think this is our next discussion.

Comment [s2]: Last week's discussion strongly suggested to me that while this component certainly exists, it will have no bearing on our notational algebra. I'm convinced Application is our highest level of concern.

Methods:

1) Method: $(name, domain, range)$
    a. We use a functional notation to consider methods as operations that have an input domain and an output range. The domain of a method is defined as the Cartesian product of its parameter types, and the range is the Cartesian product of its return types. This is the broadest definition of a method, and we will see that more specific types of methods can be expressed by adding further constraints to the domain and range. Note that this triple notation is synonymous with the classic functional notation, and will be used interchangeably.
2) Object Method (of B): $(name, B \times domain, range)$
    a. These are specified in object definitions, and are taken when invoked to apply to a single instance of that object. To express this Object-Oriented notion of instance methods, we extend our notation of methods to include the implied parameter (typed accordingly) to represent the instance on which the method is invoked. The notation of $B \times domain$ here specifies a constraint on the domain of an object method – B must be a component of the domain for any object method of B. Other components may exist as well, but are optional.
3) Constructors (of B): $(name, domain, B \times range)$
    a. Constructors for a given object are those methods, object or otherwise, whose range includes an instance of that object. The notation $B \times range$ here expresses that constraint – B must be a component of the range for any constructor of B. If a constructor is also an object method, the instance returned may be either the original instance, in a possibly modified state, or a new instance entirely. The distinction is dependent on the semantics of the method, and is beyond the scope of our notation. Common examples of Constructors include classic "setter" methods, and of course the traditional instantiating methods by that name.
4) Interrogators (of B): $(name, B \times domain, range - B)$
    a. Interrogators for a given object are those methods, instance or otherwise, whose domain includes an instance of that object but whose range does **not**. These methods may provide information about an existing instance, but cannot be used to produce or alter those instances. The classic "getters" are common examples of this type of method.
5) Calculators:
    a. Calculators are another notional classification of methods, but at this point we can see that they are redundant. Every calculator either changes the state of an instance (and is therefore a Constructor) or does not (and is therefore an Interrogator). No extra notation is needed to describe this class of methods.

Commentary on Components:

1)

2)

3)

4) Properties are specified in terms of a name and a type, where name is a String and type specifies the type of values that instances of this property are allowed to take.

5) An object (class) is specified in terms of its name, a set of property specifications, and a set of method specifications. The name is a String, and either or both of the sets of properties and methods can be empty.

    a) An object instance is specified in terms of its id, its type, and its property instances. The type must correspond to a specified object (class), and the property instances in turn correspond to the property specifications included in that object definition.

6) See explanation of Method types on page 2.

    a)

7) Next in the progression is the notion of a key. Keys are properties that, across the set of all *instances* of a given *object* (class), have a unique value.

8) It now makes sense to refer to *keyed objects*, a special subset of *objects* for which there are defined at least one key.

    a) Keyed object instances are, as expected, instances of an object (class) that satisfies the keyed object requirement. One of the property instances included in this object instance must then be an instance of the key.

9) Applications are specified in terms of a name, a set of objects, a set of relationships between those objects, and a set of methods.

Open Questions:

What is an object method instance?

Do we still have a notion of Application instance that needs to be expressed?