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 $type_i$.

Universe Sets:

1) The following notation will be used to express “universe sets” for some components. The components that will make up these sets will be defined in the progression that follows, but these sets are collected here for ease of reference.

Set	Definition	Description
$\mathbb{T}$	$\{integer, boolean, string, \ldots, object^{*}\}$	All types.
$\mathbb{V}_t$	$\{v : type_t(v) = true\}$	All values of type t.
$\mathbb{M}$	$\{m : m \text{ is a method}\}$	All methods.
$\mathbb{B}$	$\{b : b \text{ is an object}\}$	All objects.
$\mathbb{O}$	$\{o : o \text{ is an object and has a key}\}$	All keyed objects.
$\mathbf{P}$	$\{p : p \text{ is a property}\}$	All properties.
$\mathbb{X}_A$	$\left\{\{b_i \times b_j\} : b_i, b_j \in A.objs\right\}$	All binary relations

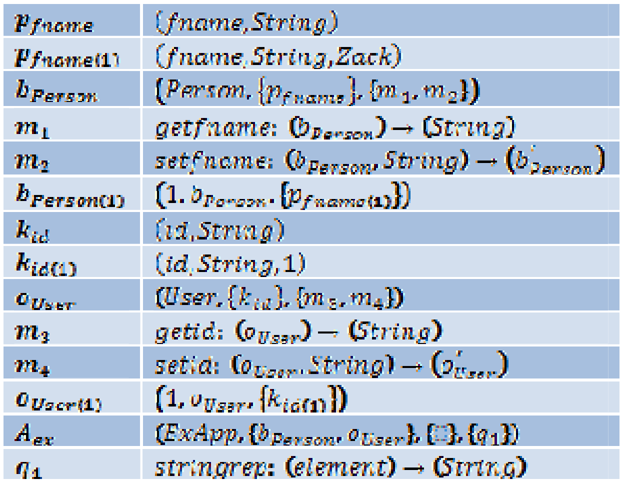
Note: $\mathbb{O} \subseteq \mathbb{B} \subset \mathbb{T}$

Progression of Components:

1) type $\in \mathbb{T}$
2) Datum (d): $(\llbracket type \rrbracket_{\downarrow i}, value) \mid \llbracket type \rrbracket_{\downarrow i}(value) = true$
3) Method (q_{name}): $f(type^{+}) \rightarrow datum^{+}$
4) Property (p_{name}): $(name, type) \mid \llbracket type \rrbracket_{\downarrow String}(name) - true$
    a) Property instance $(p_{name(id)})$:

    $(\llbracket name \rrbracket_{\downarrow i}, \llbracket type \rrbracket_{\downarrow i}, \llbracket value \rrbracket_{\downarrow i}) \mid \llbracket type \rrbracket_{\downarrow i}(\llbracket value \rrbracket_{\downarrow i}) - true$
5) Object (b_{name}) : $(name, props, mtds) \mid props \subseteq \mathbb{P}, mtds \subseteq \mathbb{M}$
    a) Object instance $(b_{\downarrow}name(id))$: $(id, type, \{p_{\downarrow}ij^{\uparrow} *\}) \mid type \in \mathbb{B}$
6) Object Method (m_{name}): $f(b, type^{*}) \rightarrow (b', datum^{*})$
7) Key (k_{name}) : $k - (p \mid \forall p_i, p_j : value_i - value_j \rightarrow i - j)$
8) Keyed Object (o_{name}): $b : \exists k \in b.props$
    a) Keyed object instance $(o_{name(id)})$: $(id, type, \{k_i, p^{*}_{ij}\})$
9) Application (A_{name}) : $(name, objs, rels, mtds) \mid objs \subseteq \mathbb{B}, rels \subseteq \mathbb{X}_{\downarrow}A$
10) Container

Examples of Components:



p_{fname}	$(fname, String)$
$p_{fname(1)}$	$(fname, String, Zack)$
b_{Person}	$(Person, \{p_{fname}\}, \{m_1, m_2\})$
m_1	$getfname: (b_{Person}) \rightarrow (String)$
m_2	$setfname: (b_{Person}, String) \rightarrow (b'_{Person})$
$b_{Person(1)}$	$(1, b_{Person}, \{p_{fname(1)}\})$
k_{id}	$(id, String)$
$k_{id(1)}$	$(id, String, 1)$
o_{User}	$(User, \{k_{id}\}, \{m_3, m_4\})$
m_3	$getid: (o_{User}) \rightarrow (String)$
m_4	$setid: (o_{User}, String) \rightarrow (o'_{User})$
$o_{User(1)}$	$(1, o_{User}, \{k_{id(1)}\})$
A_{ex}	$(ExApp, \{b_{Person}, o_{User}\}, \{\square\}, \{q_1\})$
q_1	$stringrep: (element) \rightarrow (String)$

Unary Operators:

Operator	Applies To	Notation	Example (using above)
element at	*set*	*set*[*x*]	{a, b, c}[0] = a
extractor		.	
	m	*m*.name	m_1.name = “getfname”
	d	*d*.value	(int, 2).value = 2
	d	*d*.type	(int, 2).type = int
	p	*p*.name	p_{fname}.name = “fname”
	p	*p*.type	p_{fname}.type = String
	p_i	p_i.name	$p_{fname(1)}$.name = “fname”
	p_i	p_i.value	$p_{fname(1)}$.value = “Zack”
	B	*B*.name	B_{Person}.name = “Person”
	B	*B*.props	B_{Person}.props = $\{p_{fname}\}$
	B	*B*.mtds	B_{Person}.mtds = $\{m_1,m_2\}$
	B_i	B_i.id	$B_{Person(1)}$.id = 1
	B_i	B_i.type	$B_{Person(1)}$.type = B_{Person}
	A	*A*.name	A_{ex}.name = “ExApp”
	A	*A*.mtds	A_{ex}.mtds = $\{q_1\}$
	A	*A*.objs	A_{ex}.objs = $\{B_{Person},o_{User}\}$
key of	*o*	key(*o*)	key(o_{User}) = k_{id}

Comment [s1]: Assumes keyed objects have only one key… this is an acceptable simplification!

Binary Operators:

Operator	Expression	Arg 1	Arg 2	Produces
extension	$X \obar Y$	*b*	*{m*}*	$b: (X.name, X.props, X.mtds \cup Y)$
		b	*{p*}*	$b: (X.name, X.props \cup Y, X.mtds)$
		b	*k*	$b: (X.name, X.props \cup Y, X.mtds)$
		A	*{m*}*	$A: (X.name, X.objs, X.rels, X.mtds \cup Y)$
		A	*{b*}*	$A: (X.name, X.objs \cup Y, X.rels, X.mtds)$
		A	$\sqsubseteq \mathbb{X}_x$	$A: (X.name, X.objs, X.rels \cup Y, X.mtds)$
reduction	$X \ominus Y$	*b*	*{m*}*	$b: (X.name, X.props, X.mtds - Y)$
		b	*{p*}*	$b: (X.name, X.props - Y, X.mtds)$
		A	*{m*}*	$A: (X.name, X.objs, X.rels, X.mtds \quad Y)$
		A	*{b*}*	$A: (X.name, X.objs - Y, X.rels, X.mtds)$
		A	$\subseteq \mathbb{X}_x$	$A: (X.name, X.objs, X.rels - Y, X.mtds)$
r-extension	$X \otimes Y$	*A*	*{b*}*	*A:* $X \obar Y$ *, with relations involving Y added.*
r-reduction	$\frac{X}{Y}$	*A*	*{b*}*	*A:* $X \ominus Y$ *, with relations involving Y removed.*
compose	$X + Y$	*A*	*A*	$A: (((X \oplus Y.objs) \oplus Y.rels) \oplus Y.mtds)$
r-compose	$X * Y$	*A*	*A*	$A: (((X \otimes Y.objs) \obar Y.rels) \obar Y.mtds)$
compose	$X + Y$	*A*	*A*	$A: (name', X.objs \cup Y.objs, X.rels \cup Y.rels, X.mtds \cup Y.$
r-compose	$X * Y$	*A*	*A*	$A: (name', X.objs \cup Y.objs, X.rels \cup Y.rels \cup (X.objs \times$

Ternary Operators:

Operator	Expression	Arg 1	Arg 2	Arg3	Produces
internal extend	$X\{Y \obar Z\}$	*A*	*b*	*	X, whose Y has been replaced by Y$\oplus$ Z
internal reduce	$X\{Y \ominus Z\}$	*A*	*b*	*	X, whose Y has been replaced by Y$\ominus$ Z

Basic Operations:

1) Extraction: $X.Y$
    a. Intuition: This operation is used to "extract" desired information about a component from the progression above.
    b. Input: X, Y where X is a component from the progression above and Y is the name of one field of X.
    c. Returns: Z, which corresponds to a member of X, and whose type is dependent on Y.
2) Extension: $X \oplus Y$
    a. Intuition: This operation "extends" X by adding to it the properties, objects, methods, or relations contained in Y.
    b. Input: X, Y where X is a component from the progression above and Y is a homogenous set of components that will become members of X.
    c. Returns: Z, of the same type of X where the components of Z are the union of the components of X with the elements of Y.
3) Reduction: $X \ominus Y$
    a. Intuition: This operation "reduces" X by removing from it the properties, objects, methods, or relations contained in Y. **Note: If objects are removed from X in this fashion, the possibility exists that invalid relations involving those objects are still present.**
    b. Input:
    c. Returns:
4) R-Extension: $X \otimes Y$
    a. Intuition: This operation "r-extends" X by first adding to it the objects given in Y, and then further generates new relations between each object originally in X and each object newly added from Y.
    b. Input:
    c. Returns:
5) R-Reduction: $\frac{X}{Y}$
    a. Intuition: This operation "r-reduces" X by removing from it the objects given in Y, and also removes from X any relations that involved at least one object in Y.
    b. Input:
    c. Returns:
6) Composition: $X \mid Y$
    a. Intuition:
    b. Input:
    c. Returns:
7) R-Composition: $X \bullet Y$
    a. Intuition:
    b. Input:

c. Returns:

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.

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:

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