A Categorical Model for Context

Tobias Berka
tberka@salzburgresearch.at

Contents

CHAPTER 1

Notions and Notation

1. Introduction

In this paper, we widely follow common mathematical notation. We use standard symbols for basic set expressions.

(1) $\emptyset$ ... *the empty set.*
(2) $a \in A$ ... *a is an element of the set A.*
(3) $A \subset Arrows$ ... *the set A is a subset of the set Arrows.*
(4) $A \subseteq Arrows$ ... *the set A is a subset of, or equal to, the set Arrows.*
(5) $A = Arrows$ ... *the set A is equal to the set Arrows.*
(6) $\|A\| = n$ ... *the set A consists of n elements (cardinality).*
(7) $\mathcal{P}(A)$ ... *the powerset of A.*

Function definitions also follow the common form.

$$(8) \qquad \textit{(Domain Definition)} \quad f : A \to B$$
$$(9) \qquad \textit{(Mapping Instruction)} \quad a \mapsto b,$$

where every mapping instruction can either be defined componentwise,

$$(10) \qquad a_1 \mapsto b_1$$
$$\ldots$$
$$(11) \qquad a_n \mapsto b_m,$$

or using an expression,

$$(12) \qquad a \mapsto expr(a),$$

where *expr* is any expression with a variable a (regardless of the structure of a).

We will define predicates mostly as relations,

$$(13) \qquad \begin{aligned} XOR &\subseteq \{0,1\} \times \{0,1\} \\ a\,XOR\,b &\Leftrightarrow a \wedge \neg b \vee \neg a \wedge b, \end{aligned}$$

with the typical rule for infix operators,

$$(14) \qquad aRb \Leftrightarrow R(a,b),$$

and use the most common form of logical notation in mathematics,

$$\forall \quad \textit{for all}, \tag{15}$$

$$\exists \quad \textit{exists}, \tag{16}$$

$$\exists! \quad \textit{exists exactly one}, \tag{17}$$

$$\wedge \quad \textit{and}, \tag{18}$$

$$\vee \quad \textit{or}. \tag{19}$$

Instead of using the common mathematical phrase *"... if and only if ..."*, we will simply state *"... iff ..."*. In addition to this shorthand phrase, we will use several shorthand notations, which are presented below.

2. Set Theory

Our basic notation for sets introduces several shorthand forms, which are very helpful for this paper, as we rely mostly on sets and functions as basic units of modelling.

2.1. Short Set Notation. In this paper we are using a shorthand notation to denote a *set of elements of a certain type of objects*.

The expression $\lceil A \rfloor$ denotes a set of elements of equal type as an element A. This is merely a shorthand notation for brevity and better readability of this paper.

We can thus assume that $\lceil Arrow \rfloor$ denotes a set of arrows.

2.2. Short Finite Set Notation. Our shorthand notation for sets does not make any assumptions about the cardinality of any $\lceil A \rfloor$. For our goal of describing systems, we need a differentiation between *finite* and *infinite* sets. Even though this is a very trivial matter, a concise definition is hard to come by.

To find a formal definition, we need an auxiliary function rem, which removes *exactly one* arbitrary element from a set.

$$\begin{aligned} rem : \lceil A \rfloor &\rightarrow \lceil A \rfloor \\ \{a_1, ..., a_n\} &\mapsto \{a_1, ..., a_{n-1}\} \\ \{a_1\} &\mapsto \emptyset. \end{aligned} \tag{20}$$

Using this function, we can define the notion of a *finite set*, with a corresponding shorthand notation.

Definition 2.1 (Finite Sets). We define the notion of a finite set with elements of equal type as an element A as follows.

$\lceil A \rceil$ is a finite set with elements of type A iff the following holds:

$$\exists n \in N : \lceil A \rceil \subseteq \lceil A \rfloor \wedge rem^n(\lceil A \rceil) = \emptyset, \tag{21}$$

where N is the set of natural numbers and rem^n corresponds to applying rem n times.

This symbol collides with the common notation for the upper bound for real numbers $\lceil x \rceil$, but we will limit the usage of this symbol to *our* definition of finite sets, as we do not resort to calculus in this paper.

2.3. Indexed Sets. We will use the notion of an indexed set, which is a tuple of a set, and a numbering or indexing function. This notion is a very convenient way of operating on varying numbers of ordered items from an underlying set.

DEFINITION 2.2 (Indexed Sets). An indexed set is a tuple (Set, i) of a set and a function i, which assigns a *different* integer number to every element in the set. We state $\langle Set \rangle$ to denote an indexed set.

$$\langle Set \rangle = (Set, i) \tag{22}$$

$$i \in \lceil Set \to \{1, 2, ..., \|Set\|\}\rfloor, \tag{23}$$

where i is a bijection,

$$\textit{(i injective)} \qquad \forall s, s' \in Set : i(s) = i(s') \Rightarrow s = s' \tag{24}$$

$$\wedge$$

$$\textit{(i surjective)} \quad \forall n \in \{1, 2, ..., \|Set\|\} \exists s \in Set : i(s) = n, \tag{25}$$

and $\langle Set \rangle.i$ is used to identify the indexing function of a particular set, to avoid ambiguity.

We can assume the following about $\langle Set \rangle$:

$$\|\langle Set \rangle\| := \|Set\| \tag{26}$$

$$x \in \langle Set \rangle \Leftrightarrow x \in Set \tag{27}$$

$$\langle A \rangle \subset \langle B \rangle \Leftrightarrow A \subset B \tag{28}$$

$$\langle A \rangle = \langle B \rangle \Leftrightarrow \forall a \in A \exists b \in B : a = b \wedge \langle A \rangle.i(a) = \langle B \rangle.i(b). \tag{29}$$

Let us now describe the index function i by establishing a conntection to n-tuples of elements of the underlying set. We initially define a function $\iota\tau$ for mapping in indexed set of a given length to an n-tuple.

$$\iota\tau : \langle Set \rangle \to Set^n$$
$$\big(\{s_1, ..., s_n\}, i\big) \mapsto (t_1, ..., t_n), \tag{30}$$

such that the following holds for $(t_1, ..., t_n)$:

$$\forall 1 \leq k \leq n : t_k = s \in \{s_1, ..., s_n\} \wedge i(s) = k. \tag{31}$$

We also define a similar function $\tau\iota$ from n-tuples to indexed sets.

$$\tau\iota : Set^n \to \langle Set \rangle$$
$$(t_1, ..., t_n) \mapsto \big(\{s_1, ..., s_n\}, i\big), \tag{32}$$

such that the following holds for i and $\{s_1, ..., s_n\}$:

$$\forall 1 \leq k \leq n : s_k = t_k \wedge i(s) = k. \tag{33}$$

We now have introduced the basic notion if indexed sets, and can make several additional assumptions about their properties.

DEFINITION 2.3 (Well Formed Indexed Sets). A well formed indexed set is an indexed set, for which the following holds, assuming that $id_{\langle Set \rangle}$ is the identity function for indexed sets:

$$\tau\iota \circ \iota\tau = id_{\langle Set \rangle} \Leftrightarrow \tau\iota = \iota\tau^{-1}. \tag{34}$$

We can also state several additional operators for indexed sets.

Definition 2.4 (Indexed Set Operations). First, we define three operations on indexed sets, namely for joining two indexed sets by appending one to the other, for removing all elements of one indexed sets from another and for applying functions to indexed sets.

$$\text{(35)} \qquad \textit{(Joining)} \quad \langle A\rangle + \langle B\rangle = \tau\iota(\,(\iota\tau A, \iota\tau B)\,),$$

$$\text{(36)} \qquad \textit{(Removing)} \quad \langle A\rangle - \langle B\rangle = \langle A \setminus B\rangle$$
$$\textit{so that } \langle A \setminus B\rangle \textit{ is well formed},$$

$$\text{(37)} \qquad \textit{(Functions)} \quad f(\langle A\rangle) = \langle X\rangle,\ X = \{f(a)|a \in A\} \textit{ iff } f : A \to X \textit{ or}$$
$$f(\langle A\rangle) = \langle X\rangle,\ X = \{f(A)\} \qquad \textit{iff } f : \lceil A\rfloor \to X,$$

assuming the following for the $-$ operator:

$$\text{(38)} \qquad \langle A\rangle - B \quad = \quad \langle A\rangle - \langle B\rangle.$$

Secondly, we now present several notations for accessing indexed sets.

$$\text{(39)} \qquad \textit{(Indexed Element)}\ \langle A\rangle_k \quad = \quad a_k \in \langle A\rangle,$$
$$i(a_k) = k,$$

$$\text{(40)} \qquad \textit{(Indexed Subset)}\ \langle A\rangle_{k,l} \quad = \quad \tau\iota(\,(a_k, ..., a_l)\,) \subset \langle A\rangle,$$
$$\forall k \leq h \leq l : i(a_h) = h.$$

2.4. Set Multiplication. We use an additional shorthand notation for adding a set to itself multiple times. This is very convenient if we require multiple occurances of every element of a base set.

Definition 2.5. Set multiplication is an operation on a natural number and a set, yielding a set.

$$\text{(41)} \qquad nA = \textstyle\bigcup_{i=1}^{n} A.$$

As an example, let us consider a set,

$$\text{(42)} \qquad \{A, B, C\}.$$

A set multiplication by 2, 3 or 4 would then yield,

$$\text{(43)} \qquad \begin{aligned} 2\{A, B, C\} &= \{A, A, B, B, C, C\} \\ 3\{A, B, C\} &= \{A, A, A, B, B, B, C, C, C\} \\ 4\{A, B, C\} &= \{A, A, A, A, B, B, B, B, C, C, C, C\}. \end{aligned}$$

2.5. Strings as Indexed Sequence Sets. Based on our previous definitions, we can now consider how to craft strings. We will use indexed sets as basic structure for strings.

Assuming that we have a set $\sum$ as an alphabet, we can describe a string as,

$$\text{(44)} \qquad \langle Word\rangle,\ Word \in \mathcal{P}\Big(n\sum\Big).$$

We know that the powerset is a family of sets,

$$\text{(45)} \qquad \lceil Word\rfloor \subseteq \mathcal{P}\Big(n\sum\Big),\ Word \subseteq n\sum$$

$$\text{(46)} \qquad \mathcal{P}\Big(n\sum\Big) = \lceil Set\rfloor.$$

If we now assume that $\lfloor X \rfloor$ denotes *any one element* of the *set* X, we can obtain a much shorter definition of a string:

$$\left\langle \left\lfloor \mathcal{P}\left(n \sum\right) \right\rfloor \right\rangle. \tag{47}$$

We can now define our set element shorthand notation, and present a final notation for strings, expressed as indexed sequence sets.

Definition 2.6 (Set Element Notation). We use a notation of the form $\lfloor X \rfloor$ to denote any single element of a set X.

$$x = \lfloor X \rfloor \Leftrightarrow x \in X. \tag{48}$$

Definition 2.7 (Indexed Sequence Sets). We define the notion of an indexed sequence set as an indexed set, which consists of a number of elements of a set A.

$$\left\langle {}^{n}A \right\rangle \Leftrightarrow \left\langle \left\lfloor \mathcal{P}\left(nA\right) \right\rfloor \right\rangle. \tag{49}$$

We can now describe a string of an alphabet $\sum$ and length n as,

$$\left\langle {}^{n}\sum \right\rangle, \tag{50}$$

and the set of all strings with length n as,

$$\left\lceil \left\langle {}^{n}\sum \right\rangle \right\rfloor, \tag{51}$$

3. Function Theory

For functions, we use two different notations for the domain definition:

$$f: \quad A \longrightarrow B \tag{52}$$

$$or$$

$$A \xrightarrow{f} B\ , \tag{53}$$

and assume that the inverse of a function f is denoted by f^{-1} and that a function composition is denoted by $\circ$:

$$(g \circ f)(x) = g(f(x)). \tag{54}$$

As we are using category theory, we will use the same notation for morphisms in general. For describing systems, we will introduce an additional notion for collecting functions according to the general structure of the domain definition.

3.1. Function Templates. We introduce the notion of a *function template*, which is a general set of functions according to the structure of the domain definition, but no information about the concise nature of the mapping.

Definition 3.1 (Function Templates). A function template is a set of functions, specified in the form:

$$Template = \lceil Domain \longrightarrow Image \rfloor. \tag{55}$$

These sets of functions can be used as structural tokens for member functions.

3.2. Applying Functions to Sets. For convenience, we will introduce the concept of applying a function to a set of elements, which is a subset or equal to the domain, by applying it to all elements of the set.

DEFINITION 3.2 (Applying Functions to Sets of the Domain). If we have a set $\lceil A \rfloor$ and a function from the template $f : \lceil A \rfloor \to X$, for any target domain X, we define the following rule for applying the function to the set:

$$f(\lceil A \rfloor) = f(\{a_1, ..., a_n\}) = \big\{f(a_1), ..., f(a_n)\big\}. \tag{56}$$

4. Category Theory

A category is a mathematical construct, which follows certain rules (see [**5**]). As such, a given construct must meet certain requirements to be a category. We will now present the definition of a category.

DEFINITION 4.1 (Categories). A category consists of a class of objects and morphisms between these objects. We describe the class of objects as,

$$Objects \quad = \quad \lceil O \rfloor, \tag{57}$$

without any assumptions about the nature of all $o \in \lceil O \rfloor$.

Any morphisms exists between two objects, and may have further characteristics. We hence state,

$$Morphisms \quad = \quad Objects \times Objects \times X, \tag{58}$$

where X is a set, or a cartesian product of sets, for all further characteristics beyond the source and target of a morphism.

We can now describe the *set of all categories*, which serves as a structural description of categories.

$$\lceil Category \rfloor = \lceil Objects \rfloor \times \lceil Morphisms \rfloor. \tag{59}$$

A category is now structurally described as a tuple of objects and morphisms.

Now that we can *structurally* describe a category, we can define means of *accessing* this structure, and then present the basic *requirements* for categories.

4.1. Basic Category Theory. We will use calligraphic upper-case strings to denote categories.

$$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{ABC} \quad ... \quad categories. \tag{60}$$

When dealing with categories, we use two operators, namely $obj(\mathcal{A})$ and $mor(\mathcal{A})$, to refer to all objects or morphisms of the category $\mathcal{A}$.

DEFINITION 4.2 (Objects in a Cateogry). We can write $obj(\mathcal{A})$ to denote all objects in $\mathcal{A}$. We can define obj as a general function on categories.

$$\begin{aligned} obj : \lceil Category \rfloor \quad &\to \quad \lceil Objects \rfloor \\ (objects, morphisms) \quad &\mapsto \quad objects. \end{aligned} \tag{61}$$

DEFINITION 4.3 (Morphisms in a Category). We can write $mor(\mathcal{A})$ to denote all morphisms in $\mathcal{A}$. We can define mor as a general function on categories.

$$\begin{aligned} mor : \lceil Category \rfloor \quad &\to \quad \lceil Morphisms \rfloor \\ (objects, morphisms) \quad &\mapsto \quad morphisms. \end{aligned} \tag{62}$$

We now define a binary relation, which is also denoted by *mor*, but structurally different as the function, for checking pairs of objects for morphisms.

$$\begin{aligned} mor &\subseteq obj(\mathcal{A}) \times obj(\mathcal{A}) \\ mor(A,B) &\Leftrightarrow (A,B,X) \in mor(\mathcal{A}), \end{aligned} \tag{63}$$

where X is again the set, or a cartesian product of sets, of all further characteristics.

Now that we have an initial vocabulary, we can present the rules, which must hold for every category.

Definition 4.4 (Categorical Requirements). A category $\mathcal{C}$,

$$\mathcal{C} \in \lceil Category \rfloor, \tag{64}$$

must meet the following constraints:

- Identity: For every object, there must be an identity morphism.

$$\forall o \in obj(\mathcal{C}) : \; o \circlearrowright_{id_o} \in mor(\mathcal{C}) \tag{65}$$

- Composition: For every pair of morphisms, there must be a composite morphism,

$$\forall\ a \xrightarrow{f} b \xrightarrow{g} c\ \in mor(\mathcal{C}) : \exists\ a \xrightarrow{g \circ f} c\ \in \lceil Morphisms \rfloor, \tag{66}$$

  which meets the following requirements:
  - Associativity Law:

$$h \circ (g \circ f) = (h \circ g) \circ f. \tag{67}$$

  - Identity Law:

$$\forall\ a \xrightarrow{f} b\ \in mor(\mathcal{C}) : f \circ id_a = id_b \circ f = f. \tag{68}$$

For convenience and computability, we will also provide three functions on morphisms, which provide access to the target (image), source (preimage) and attributes (further characteristics) of every morphism.

Definition 4.5 (Accessing Morphisms). We define three functions *image*, *preimage* and *attributes* to access the key characteristics of every morphism.

$$\begin{aligned} preimage : Morphisms &\rightarrow Objects \\ (p,i,a) &\mapsto p, \end{aligned} \tag{69}$$

$$\begin{aligned} image : Morphisms &\rightarrow Objects \\ (p,i,a) &\mapsto i, \end{aligned} \tag{70}$$

$$\begin{aligned} attributes : Morphisms &\rightarrow X \\ (p,i,a) &\mapsto a, \end{aligned} \tag{71}$$

where X is the set, or a cartesian product of sets, of all further characteristics.

4.2. Example of a Category. Our categories will be very similar to a classical example of a category, the category of arrows, which we will revisit briefly.

The category of arrows consists of abstract objects.

$$obj(\mathcal{A}) = \lceil Objects \rfloor. \tag{72}$$

We have a set of arrows, which are defined over the objects, between they exist.

$$mor(\mathcal{A}) = \lceil Objects \rfloor \times \lceil Objects \rfloor, \tag{73}$$

$$(A, B) \in mor(\mathcal{A}) \Leftrightarrow A \to B. \tag{74}$$

This means that our morphisms have no attributes in this category.

$$X = \emptyset. \tag{75}$$

Every object has the identity arrow:

$$\forall A \in obj(\mathcal{A}) : A \to A \in mor(\mathcal{A}). \tag{76}$$

The composition rule for arrows is as follows:

$$A \to C \ \textit{iff}\ A \to B \wedge B \to C. \tag{77}$$

The categorical requirements are easily verified for the category of arrows.

4.3. Strong and Weak Functors. The notion of a weak functor refers to an assignment between categories. This assignment is characterized by two sets of mappings, one for objects and another for morphisms. Based on our structural description of a category, we now define the notion of a weak functor.

Definition 4.6 (Weak Functors). A weak functor is an assignment between categories, with mappings for objects and morphisms. This means that a functor is characterized by a triple,

$$\lceil Functor \rfloor = \lceil \mathcal{A} \to \mathcal{B} \rfloor \times \left\lceil \lceil A \mapsto B \rfloor \right\rfloor \times \left\lceil \lceil \alpha \hookrightarrow \beta \rfloor \right\rfloor \tag{78}$$

$$F = (domain\ definition, map_{obj}, map_{mor}) \in \lceil Functor \rfloor, \tag{79}$$

for which the following must hold:

$$\mathcal{A}, \mathcal{B} \in \lceil Category \rfloor \qquad \wedge \tag{80}$$

$$\wedge \quad A \in obj(\mathcal{A}) \wedge B \in obj(\mathcal{B}) \wedge \alpha \in mor(\mathcal{A}) \wedge \beta \in mor(\mathcal{B}).$$

To define and describe functors, we will use the following notation:

$$F = (\mathcal{A} \to \mathcal{B}, \lceil A \mapsto B \rfloor, \lceil \alpha \hookrightarrow \beta \rfloor) \tag{81}$$

$$\Leftrightarrow$$

$$\mathcal{A} \longrightarrow \mathcal{B} \tag{82}$$

$$a \longmapsto b \tag{83}$$

$$\phi \hookrightarrow \psi,$$

where the $\hookrightarrow$ is also a mapping (like $\mapsto$), but with a different arrow for readability.

We can present three rules for applying a general functor to an object, a morphism or an entire category. Let $F \in \lceil Functor \rfloor$ denote a weak functor. The rules for applying a functor are:

$$\forall X \in Objects \ : \ F(X) = Y, \ such\ that\ X \mapsto Y \in map_{obj} \tag{84}$$

$$\forall \phi \in Morphisms \ : \ F(\phi) = \psi, \ such\ that\ \phi \to \psi \in map_{mor} \tag{85}$$

$$\forall \mathcal{A} \in \lceil Category \rfloor \ : \ F(\mathcal{A}) = (\, F(objects_{\mathcal{A}}), F(morphisms_{\mathcal{A}}) \,). \tag{86}$$

Weak functors do not have any properties or requirements beyond our definition, but we will now introduce the notion of a *strong* functor, which we will refer to simply as a *functor*, for shortness and compliance with the common notation.

Definition 4.7 ((Strong) Functors). A (strong) functor $F : \mathcal{A} \to \mathcal{B}$, simply called *functor*, is a weak functor, which meets the following requirement:

$$(87) \forall f : a \to b \in mor(\mathcal{A}) : F(f : a \to b) = F(f) : F(a) \to F(b) \in mor(\mathcal{B}).$$

We can consider functors to be *structure preserving* mappings, as they reflect the connectivity of the source category in the target category. Conversely, we can assume that weak functors are *not* structure preserving mappings.

CHAPTER 2

Basic Context Categories

1. Methods, Objects and Abstract Context Entities

In this section we will make some basic considerations about the layer of abstract context entities. To do so, we need several basic notions, which can or must be kept open and abstract.

If we refer to objects, we simply assume that these objects are *not* context descriptions, but we do not make any further assumptions beyond the fact that they exist. As an initial means of having different *types* of objects, we will use a notion independent from data types, that of object classes.

Definition 1.1 (Object Classes). An object class is simply a collection of objects.

$$\lceil ObjectClass \rfloor \subseteq \left\lceil \lceil Object \rfloor \right\rfloor. \tag{88}$$

In a closed system, the classes of objects should form a partition of the system objects.

$$\forall o \in SystemObjects \subseteq \lceil Object \rceil \exists! C \in SystemClasses \subseteq \lceil ObjectClass \rceil : o \in C, \tag{89}$$

meaning that every object from a finite set of *objects in the system* must be a member of exactly one object class from a finite set of *object classes in the system* (see Chapter 1 for details on our notation).

Type definitions can be used to form object classes by simply collecting their instances in a set. We now need the notion of a method to define our abstract context entities.

1.1. Methods Create Context. A particular description of context is always created by a method. If order to gain a flexible and open definition, we will define these over the set of all methods, $\lceil Method \rfloor$.

Definition 1.2 (Methods). If X is any source, from which a context may be created, the set of all methods is defined as,

$$\lceil Method \rfloor = \left\lceil X \to \lceil Context \rfloor \right\rfloor, \tag{90}$$

This statement defines methods according to a basic "one-sentence principle", namely, "methods create context", while still leaving the source of information necessary to create a context open.

At the moment we are only concerned with defining our basic modelling of context and morphisms on context, but this definition can be used to provide an interface between objects and the actual computation of contexts at a later stage.

1.2. Context as Abstract Mathematical Entities. We will define an abstract context entity as an abstract object, without making assumptions on the internal structure or nature of this object (according to [**3**]). Instead, we will define abstract context entities based on a similar structure as morphisms.

Definition 1.3 (Abstract Context Entities). A set of abstract context entities is defined over the method, the source objects and the view object class. By describing only the external connections and underlying method, they form tokens for one or many internal representations.

$$\lceil Context \rfloor = \lceil Method \rfloor \times \left\lceil \lceil Object \rfloor \right\rfloor \times \lceil ObjectClass \rfloor. \tag{91}$$

We make no assumptions regarding the internal structure.

2. Context Types and Groups

As the next step, we will define our notion of context types and context groups. These will allow us to gain further insights at a later stage, and will be an important part of describing systems which use different notions of context.

2.1. Context Types. We will differentiate between different *types* of context, based on what class of objects this context describes, what aspect is being considered and by which method it is generated. These descriptions serve as black-box type definitions for our abstract context entities.

Definition 2.1 (Context Types). A context type τ_c is defined as a triple of the creating method, the source object class and the view object class.

$$\lceil ContextType \rfloor = \lceil Method \rfloor \times \lceil ObjectClass \rfloor \times \lceil ObjectClass \rfloor, \tag{92}$$

$$\tau_c = (creating\,method, object, view\,object\,class) \in \lceil ContextType \rfloor. \tag{93}$$

This definition is structurally almost identical to our general structure of a morphism (see Section 4 in Chapter 1),

$$Morphisms = Objects \times Objects \times X. \tag{94}$$

The structural similarity reflects the consideration that context types can be used to describe functional dependencies, which are also candidates for a morphism representation.

2.2. Context Groups. If we disregard the methods, which create a context, we come to the more general notion of context groups. These context groups collect different context types based purely on the classes of objects, which describe the respective context types.

Definition 2.2 (Context Groups). We will now define the notion of a context group γ_c, as a tuple of the source and view object class.

$$\lceil ContextGroup \rfloor = \lceil ObjectClass \rfloor \times \lceil ObjectClass \rfloor \tag{95}$$

$$\gamma_c = (object\,class, view\,object\,class). \tag{96}$$

A classical interpretation of context groups would describe them as “unlabeled many-to-many functional dependencies”.

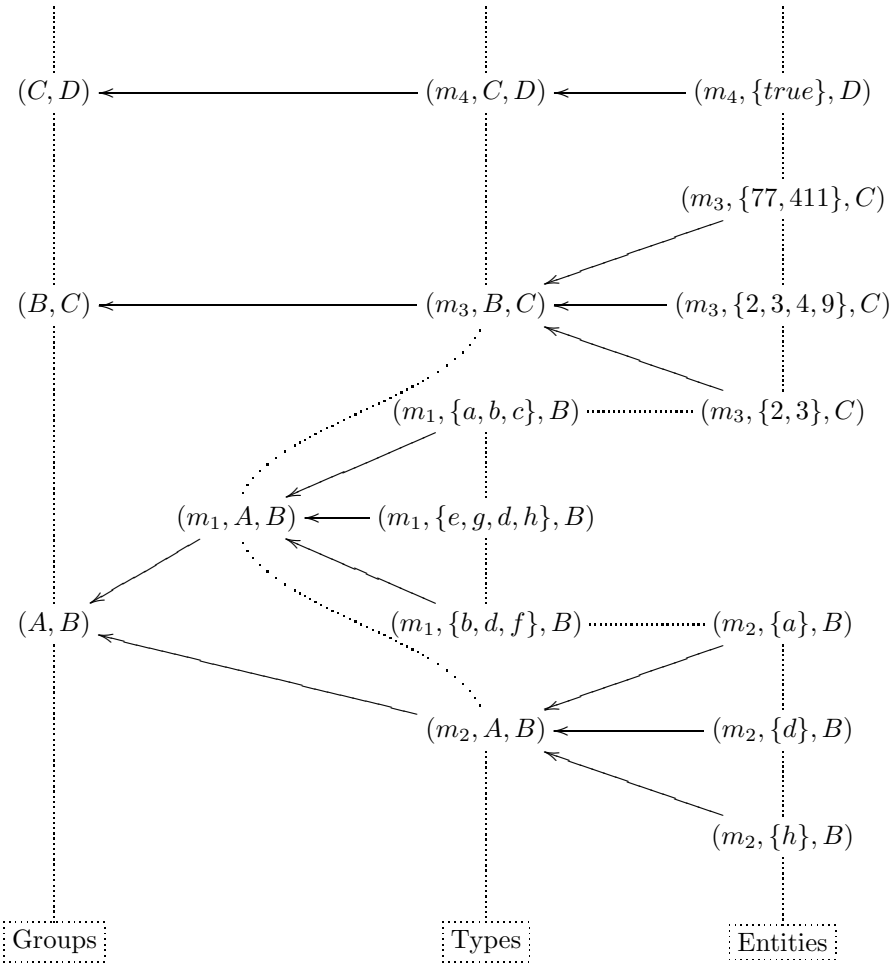

FIGURE 1. Context Groups, Types and Entities

2.3. Function-Based Mappings between Context Notions. With the above definitions, we will propose some initial rules for the basic modeling of context.

First we will define a mapping from abstract context entities to context types, to model the fact that context entities are instances of context types. As in most other models, this leads to a one-to-many assignment from types to entities.

We will then define a mapping from context types to context groups, representing the idea that a context group can "collect" multiple context types by disregarding the creating methods, thus loosing some semantics, but serving as a generalization. Figure 1 depicts the resulting mappings between entities, types and groups

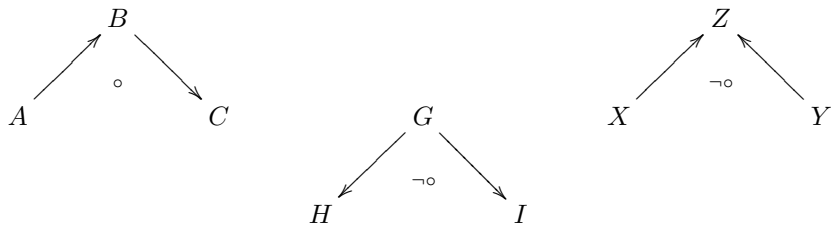

FIGURE 2. Context Group Connectivity

DEFINITION 2.3 (Mapping from Abstract Context Entities to Context Types). We define a function T from context entities to context types.

$$\begin{aligned} T : \lceil Context \rfloor &\rightarrow \lceil ContextType \rfloor \\ (m, O, C_v) &\mapsto (m, C_o, C_v) \text{ such that } O \subseteq C_o. \end{aligned} \tag{97}$$

DEFINITION 2.4 (Mapping from Context Types to Context Groups). Furthermore, we now define a function Γ from context types to a context groups.

$$\begin{aligned} \Gamma : \lceil ContextType \rfloor &\rightarrow \lceil ContextGroup \rfloor \\ (m, C_o, C_v) &\mapsto (C_o, C_v). \end{aligned} \tag{98}$$

3. Context Connectivity

In order to make considerations about context connectivity, we will view the highest level, that of context groups. We will first define a rule for the connectivity amongst context groups. Then, we will translate it into the language of morphisms and apply it to our other levels of viewing context.

3.1. The Context Group Connectivity Rule. We now define a composition rule for context groups.

DEFINITION 3.1 (Composition Rule on Context Groups). We introduce the context connectivity rule $\circ$, which is defined as a binary relation over the set of context groups. Two context groups are considered to be *basically composable*, if (i) one context type's first object class matches another context type's second object class, and (ii) every context group is composable with itself.

$$\begin{aligned} \circ &\subseteq \lceil ContextGroup \rfloor \times \lceil ContextGroups \rfloor \\ (A, B) \circ (C, D) &\Leftrightarrow B = C^{(i)} \vee (A = C \wedge B = D)^{(ii)}. \end{aligned} \tag{99}$$

This does not define the concise nature of the composition, as it would by stating which result a composition would yield, leading to a connectivity as depicted in the Figure 2.

3.2. Mappings for Context Connectivity. Previously, we have defined a composition rule for context groups (see Statement 99 in Section 2.2). We will now model an analog function for context groups, and generalize it for context types and abstract context entities.

Definition 3.2 (Functional Connectivity Representation between Context Groups). We define a morphism $\circ_{CG}$, which represents all possible compositions amongst context groups, according to our definition of the basic composition rule $\circ$.

$$\begin{aligned} \circ_{CG} : \lceil ContextGroup \rfloor &\rightarrow \lceil \lceil ContextGroup \rfloor \rfloor \\ (C_i, C_j) &\mapsto \{\gamma | \gamma \circ (C_i, C_j)\}. \end{aligned} \tag{100}$$

Let us not define similar functions for context types and contexts.

Definition 3.3 (Functional Connectivity Representation between Context Types). We define a morphism $\circ_{CT}$, which represents all possible compositions amongst context types, based on $\circ_{CG}$.

$$\begin{aligned} \circ_{CT} : \lceil ContextType \rfloor &\rightarrow \lceil \lceil ContextType \rfloor \rfloor \\ \tau_c &\mapsto \{\tau_r | \Gamma(\tau_r) \in \circ_{CG}(\Gamma(\tau_c))\}. \end{aligned} \tag{101}$$

Definition 3.4 (Functional Connectivity Representation between Abstract Context Entities). The definition of the corresponding morphism $\circ_{AC}$ is similar.

$$\begin{aligned} \circ_{AC} : \lceil Context \rfloor &\rightarrow \lceil \lceil Context \rfloor \rfloor \\ c &\mapsto \{r | T(r) \in \circ_{CT}(T(c))\}. \end{aligned} \tag{102}$$

4. Categorical Modelling of Context

We will now attempt to model three categories for context groups, types and entities, denoted by $\mathcal{CG}$, $\mathcal{CT}$ and $\mathcal{AC}$, and (strong) functors for the mapping from entities to types and types to groups. This is a basic vocabulary, which we will refer to many times in the following chapters.

4.1. The Category of Context Groups.

We will now define the category of context groups.

Definition 4.1 (The Category of Context Groups). The category $\mathcal{CG}$ of context groups is defined as follows:

$$obj(\mathcal{CG}) = \lceil ContextGroup \rfloor, \tag{103}$$

$$mor(\mathcal{CG}) \subseteq \lceil ContextGroup \rfloor \times \lceil ContextGroup \rfloor, \tag{104}$$

$$\forall \gamma_1, \gamma_2 \in obj(\mathcal{CG}) : \gamma_2 \in \circ_{CG}(\gamma_1) \Leftrightarrow (\gamma_1, \gamma_2) \in mor(\mathcal{CG}), \tag{105}$$

$$\exists (A, B) \in mor(\mathcal{CG}) \Leftrightarrow A \rightarrow B, \tag{106}$$

$$\forall A, B, C \in obj(\mathcal{CG}) : A \rightarrow C \textit{ iff } A \rightarrow B \wedge B \rightarrow C \tag{107}$$

$$\forall A \in obj(\mathcal{CG}) : A \xrightarrow{id_A} A. \tag{108}$$

We can easily show that this meets all requirements to be a category.

4.2. The Category of Context Types.

We will now define the analog category of context types.

DEFINITION 4.2 (The Category of Context Types). The category $\mathcal{CT}$ is defined as follows:

$$obj(\mathcal{CT}) = \lceil ContextType \rfloor, \tag{109}$$

$$mor(\mathcal{CT}) \subseteq \lceil ContextType \rfloor \times \lceil ContextType \rfloor, \tag{110}$$

$$\forall \tau_1, \tau_2 \in obj(\mathcal{CT}) : \tau_2 \in \circ_{CT}(\tau_1) \Leftrightarrow (\tau_1, \tau_2) \in mor(\mathcal{CT}), \tag{111}$$

$$\exists (A, B) \in mor(\mathcal{CT}) \Leftrightarrow A \to B, \tag{112}$$

$$\forall A, B, C \in obj(\mathcal{CT}) : A \to C \textit{ iff } A \to B \wedge B \to C \tag{113}$$

$$\forall A \in obj(\mathcal{CT}) : A \xrightarrow{id_A} A. \tag{114}$$

It again meets all requirements to be a category.

4.3. The Category of Abstract Context Entities. We also define an analog category of abstract context entities.

DEFINITION 4.3 (The Category of Abstract Context Entities). The category $\mathcal{AC}$ is defined as follows:

$$obj(\mathcal{AC}) = \lceil Context \rfloor, \tag{115}$$

$$mor(\mathcal{AC}) \subseteq \lceil Context \rfloor \times \lceil Context \rfloor, \tag{116}$$

$$\forall c_1, c_2 \in obj(\mathcal{AC}) : c_2 \in \circ_{AC}(c_1) \Leftrightarrow (c_1, c_2) \in mor(\mathcal{AC}), \tag{117}$$

$$\exists (A, B) \in mor(\mathcal{AC}) \Leftrightarrow A \to B, \tag{118}$$

$$\forall A, B, C \in obj(\mathcal{AC}) : A \to C \textit{ iff } A \to B \wedge B \to C \tag{119}$$

$$\forall A \in obj(\mathcal{AC}) : A \xrightarrow{id_A} A. \tag{120}$$

It meets all requirements to be a category.

4.4. Functors Between $\mathcal{AC}$, $\mathcal{CT}$ and $\mathcal{CG}$. According to our mappings between contexts, context types and context groups in Section 2.3 we will now define functors between our categorical notions of context.

DEFINITION 4.4 (A Functor between $\mathcal{AC}$ and $\mathcal{CT}$). The T functor between the categories of abstract context entities and context types is defined as follows:

$$\begin{array}{rcl} \mathcal{AC} & \xrightarrow{T} & \mathcal{CT} \\ c_i & \longmapsto & T(c_i) \\ c_i \longrightarrow c_j & \hookrightarrow & T(c_i) \longrightarrow T(c_j)\ , \end{array} \tag{121}$$

where the T function used in the object mapping is the function from abstract context entities to types (see Definition 2.3).

DEFINITION 4.5 (A Functor between $\mathcal{CT}$ and $\mathcal{CG}$). The Γ functor between the categories of abstract context entities and context types is defined as follows:

$$\begin{array}{rcl} \mathcal{CT} & \xrightarrow{\Gamma} & \mathcal{CG} \\ \tau_i & \longmapsto & \Gamma(\tau_i) \\ \tau_i \longrightarrow \tau_j & \hookrightarrow & \Gamma(\tau_i) \longrightarrow \Gamma(\tau_j)\ , \end{array} \tag{122}$$

where the Γ function used in the object mapping is the function from context types to groups (see Definition 2.4).

As the connectivity of $\mathcal{AC}$ and $\mathcal{CT}$ have the same origin as that of $\mathcal{CG}$, and it is passed down based on the Γ and T function, it is easy to proove that they respect the structure and connectivity, and can thus be considered to be functors.

CHAPTER 3

Type Concept Specification

1. What Are Type Concepts?

In order to offer more functionality we now need some means of specifying the internal nature of a context type and context entities. However, it is still our goal to be application and implementation independent.

This means that we need a model for data-types with the right level of detail on different aspects. We do not want an in-depth specification of implementational data-storage, as class diagrams and UML would provide, but we do want to be able to specify some semantics about these data-types in a way, which allows us to process these descriptions in an automated manner.

But before we can consider high-level descriptions, such as semantics, we first need to define some basic notions. We will initially describe some basic types for storage, which will be called *structure types*, before adding several descriptions of their semantics. This combination of structure types and semantics will then form our notion of *type concepts*.

If we recapture typical software systems using AI, we can very often find one or more of the following data-types:

- term frequency vector
- key-term list
- numerical value
- topic rating vector
- topic-to-term mapping

These specifications are very expressive, in terms of algorithms, and still very general in terms of implementations. In order to reach our definition for type concepts, we will decompose these sample data-types, and consider them part by part.

2. Structure Types

Our first definitions will only consider that part of data types, which describe the basic *structure* of the types. To do so, we decompose a "term frequency vector" to a "vector". This means, that can extract the following *structure types* from common representations in AI, adding the structure type *matrix* for completeness:

- value
- vector
- matrix
- list
- assignment

This allows us to define the *set of structure types*.

Definition 2.1 (Structure Type Set). The set of all structure types, denoted by $\lceil StructureType \rfloor$, is defined as follows:

$$\lceil StructureType \rfloor = \{ Value_X, Vector_X, Matrix_X, List_X, Assignment \}. \tag{123}$$

All of these structure types are *typed*, meaning that they internally consist of one basic type only, and come with their own semantics. These can be described in the following way:

- A value is a single element from a numerical domain, such as the natural numbers, integers, real numbers or complex numbers, or a single element of any other basic data-types from computer sciences, such as a string or boolean value.
- A vector is a multi-dimensional value, but it must be element of a vector space. Components are accessed over indices. Note that a component of a vector is *never* another structure type.
- A matrix is an n *by* m multi-dimensional value, but it must be element of a vector space. Components are accessed over index tuples.
- A list is a variable size collection of values or objects. A list can be accessed using indices.
- An assignment is a description of how to resolve objects from one domain to another.

We can now define the basic structure types in greater detail. In order to gain mathematical definitions of our structure types, we first need a preliminary assessment. We do not want to limit the primitive types, which can be combined with our structure types. This is why we assume to have a set of primitive types, denoted by:

$$\lceil PrimitiveType \rfloor. \tag{124}$$

This set contains all atomic type definitions, which typically include integers, real valued types, boolean values and strings.

For the moment, we will assume that we are dealing with a set of primitive types and object references only. To do so, we use a definition denoted by $\lceil Type \rfloor$.

Definition 2.2 (Type Set). The set of all types $\lceil Type \rfloor$ is defined as:

$$\lceil Type \rfloor = \lceil PrimitiveType \rfloor \cup \lceil Object \rfloor, \tag{125}$$

where $o \in \lceil Object \rfloor$ can be any reference to an object.

Using this set, we can now define our structure types more concisely. We start with the quite trivial definition of a value, and then move on to the other structure types.

Definition 2.3 (Value). If we assume that $i :: T$ denotes that an instance i has type T, the structure type $Value$ is defined as follows.

$$\forall v \in \lceil Value \rfloor_X : v :: X, \tag{126}$$

where X is any type but an object reference, or mathematically:

$$X \in \lceil Type \rfloor \setminus \lceil Object \rfloor. \tag{127}$$

Next, we define the structure types vector and matrix.

DEFINITION 2.4 (Vector and Indexed Access). The structure type $Vector$ is defined as follows.

$$\lceil Vector \rfloor_X \subseteq \left\lceil \langle^n \lceil Value \rfloor_X \rangle \right\rfloor, \tag{128}$$

which means that a vector is a sequence of values, where X is any type but an object reference, or mathematically:

$$X \in \lceil Type \rfloor \setminus \lceil Object \rfloor. \tag{129}$$

We have now defined a vector as an indexed set of values (see Section 2.3 in Chapter 1). We have a function $component_{vector}$ for accessing a given component of a vector.

$$\begin{aligned} component_{vector} : \lceil Vector \rfloor_X \times N &\rightarrow \lceil Value \rfloor_X \\ (\langle V \rangle, i) &\mapsto \langle V \rangle_i. \end{aligned} \tag{130}$$

Furthermore, we have a function sub_{vector} for accessing a part of a vector.

$$\begin{aligned} sub_{vector} : \lceil Vector \rfloor_X \times N^2 &\rightarrow \lceil Value \rfloor_X \\ (\langle V \rangle, i, j) &\mapsto \langle V \rangle_{i,j}. \end{aligned} \tag{131}$$

Finalizing, we need a function dim, which returns the dimensionality of a vector.

$$\begin{aligned} dim_{vector} : \lceil Vector \rfloor_X &\rightarrow N \\ \langle V \rangle &\mapsto \|\langle V \rangle\|. \end{aligned} \tag{132}$$

DEFINITION 2.5 (Matrix and Indexed Access). The structure type $Matrix$ is defined as follows.

$$\lceil Matrix \rfloor_X \subseteq \left\lceil \langle^n \lceil Vector \rfloor_X \rangle \right\rfloor, \tag{133}$$

which means that a matrix is a sequence of vectors, where X is any type but an object reference, or mathematically:

$$X \in \lceil Type \rfloor \setminus \lceil Object \rfloor. \tag{134}$$

Note that we have defined a matrix as an indexed set of indexed sets $\left\langle \langle Set \rangle \right\rangle$, storing every matrix row in a set.

We have a function $component_{matrix}$ for accessing a given component of a matrix.

$$\begin{aligned} component_{matrix} : \lceil Matrix \rfloor_X \times N^2 &\rightarrow \lceil Value \rfloor_X \\ (\langle V \rangle, i, j) &\mapsto \left\langle \langle V \rangle_j \right\rangle_i, \end{aligned} \tag{135}$$

for any row i and column j.

We have a function row_{matrix} for accessing a given row of a matrix.

$$\begin{aligned} row_{matrix} : \lceil Matrix \rfloor_X \times N &\rightarrow \lceil Vector \rfloor_X \\ (\langle V \rangle, i) &\mapsto \left\langle \langle V \rangle \right\rangle_i. \end{aligned} \tag{136}$$

We also have a function $column_{matrix}$ for accessing a given column of a matrix.

$$\begin{aligned} column_{matrix} : \lceil Matrix \rfloor_X \times N &\rightarrow \lceil Vector \rfloor_X \\ (\langle V \rangle, i) &\mapsto \left\langle \langle V \rangle_i \right\rangle_1 + ... + \left\langle \langle V \rangle_i \right\rangle_n, \end{aligned} \tag{137}$$

which composes all row elements.

Furthermore, we have a function sub_{matrix} for accessing a given part of a matrix.

$$\begin{aligned} sub_{matrix} : \lceil Matrix \rfloor_X \times N^4 &\rightarrow \lceil Matrix \rfloor_X \\ \big(\langle V \rangle, i_1, j_1, i_2, j_2\big) &\mapsto \Big\langle \langle V \rangle_{i_1,i_2} \Big\rangle_{j_1} + ... + \Big\langle \langle V \rangle_{i_1,i_2} \Big\rangle_{j_2}, \end{aligned} \tag{138}$$

which composes all row selections.

We now define a list structure type. At this point, it is identical to a vector. The difference between a vector and a list lies in the different semantics. This difference will be clarified in Section 3.

Definition 2.6 (List and Indexed Access). The structure type $List$ is defined as follows.

$$\lceil List \rfloor_X = \lceil Vector \rfloor_X, \tag{139}$$

where X is any type:

$$X \in \lceil Type \rfloor. \tag{140}$$

We have a function $component_{list}$ for accessing a given component of a list, which is identical to that of a vector.

$$component_{list} = component_{vector}. \tag{141}$$

Analogously, we have a function sub_{list} for accessing a given component of a list.

$$sub_{list} = sub_{vector}. \tag{142}$$

Furthermore, we will define a function for appending one list to another, denoted by $append_{list}$.

$$\begin{aligned} append_{list} : \lceil List \rfloor_X \times \lceil List \rfloor_X &\rightarrow \lceil List \rfloor_X \\ \big(\langle L_1 \rangle, \langle L_2 \rangle\big) &\mapsto \langle L_1 \rangle + \langle L_2 \rangle. \end{aligned} \tag{143}$$

We also have an operator for obtaining the length of a list:

$$length_{list} = dim_{vector}. \tag{144}$$

Finally, we need to define the structure type of an $Assignment$. We obtain this structure type by defining it as a function template on abstract object classes.

Definition 2.7 (Assignments and Accessing Assignments). The structure type $Assignment$ is defined as follows:

$$\lceil Assignment \rfloor = \lceil ObjectClass \rightarrow ObjectClass' \rfloor. \tag{145}$$

The underlying sets of an assignment can be accessed over the following functions:

$$\begin{aligned} dom_{assignment} : \lceil Assignment \rfloor &\rightarrow \lceil ObjectClass \rfloor \\ (ObjectClass \rightarrow ObjectClass') &\mapsto ObjectClass, \end{aligned} \tag{146}$$

$$\begin{aligned} codom_{assignment} : \lceil Assignment \rfloor &\rightarrow \lceil ObjectClass \rfloor \\ (ObjectClass \rightarrow ObjectClass') &\mapsto ObjectClass'. \end{aligned} \tag{147}$$

The source and target value of a given assignment can be accessed using the following functions:

$$target_{assignment} : \lceil Assignment \rfloor \times ObjectClass \to ObjectClass' \quad (148)$$
$$(f, o) \mapsto f(o),$$
$$source_{assignment} : \lceil Assignment \rfloor \times ObjectClass' \to \lceil ObjectClass \rfloor \quad (149)$$
$$(f, f(o)) \mapsto \{o' | f(o') = f(o)\}.$$

Now that we have made our basic definitions about structure types, we move on to describing semantics.

3. Specifying Semantics

Going back to the initial considerations about the data-structures involved in many system designs, we recall that our samples included the following data structures:

- term-frequency vector
- topic rating vector
- key-term list

In this section, we will attempt to develop a system for describing and discriminating these different types of data. In the last section, we found that the basic "storage layout" of a vector and a list are identical in our mathematical model, and most implementations in standard libraries seem to confirm this consideration. It is our belief that the fundamental difference lies in the semantics of these types, and that modelling the main representation of data-types in such a way is of advantage for our model.

Let us now make the following assumptions about the three above data structures, which originate in standard implementation approaches:

- A term-frequency vector uses the structure type of a vector, and the individual entries are real-valued.
- A topic rating vector is identical to a term-frequency vector in terms of data storage.
- A key-term list is a list consisting of string entries.

This means that term-frequency vectors and topic rating vectors are almost identical in terms of data storage, whereas a key-term list is different, as it relies on string entries instead of numerical entries, but it is hard to argue why it should be a list rather than a vector. Let us first consider how we can create a stronger differetiation between term-frequency and topic rating vectors.

3.1. Vector Index Semantics.

A rather strong differentiation can be crafted by adding a description what the individual components *represent*. As before, we will use object classes to do so in a flexible way. This means that a term-frequency vector is "about n-grams or words" and a topic rating vector is "about topics".

This allows us to define a new notion, that of an "applied vector", which specifies that a vector's *component values* refer to object from a certain class.

Definition 3.1 (Applied Vector). An $AppliedVector$ is defined as a $Vector$ and the class of objects, which the individual components' values are based on,

refer to or have been derived from.

$$\lceil AppliedVector \rfloor_X = \lceil Vector \rfloor_X \times \lceil ObjectClass \rfloor \tag{150}$$

$$AppliedVector = (Vector_X, ObjectClass). \tag{151}$$

As a further refinement, we introduce an additional type, which allows us to specify which individual *objects* from the specified object class are involved in an individual component. We will call this new type a bound vector, as it is an even stronger specification of semantics than in an applied vector.

Definition 3.2 (Bound Vector). A bound vector is an extension to an applied vector. For all components, it also provides the object from the object class, which is involved in every individual component. These objects are collected in a list.

A $BoundVector$ is defined as follows.

$$\lceil BoundVector \rfloor_X = \lceil Vector \rfloor_X \times \lceil ObjectClass \rfloor \times \lceil List \rfloor_{\lceil Object \rfloor} \tag{152}$$

$$BoundVector_X = (Vector_X, ObjectClass, IndexObjects_{\lceil Object \rfloor}). \tag{153}$$

Furthermore, if we have a relation $\rightsquigarrow$, expressing that an object is involved with a specific component, the following must holds:

$$\begin{aligned} component_{vector}(Vector, i) &\rightsquigarrow component_{list}(IndexObjects, j) \\ &\Rightarrow \\ i &= j. \end{aligned} \tag{154}$$

We can now specify a term-frequency vector by creating an applied vector, using $Words$ as the object class. But we can also be even more specific, as a bound vector can be created for a specific vocabulary, simply by adding the individual words to the $IndexObjects$, with indices matching those of the basic vector.

3.2. Matrix Column-Row Index Semantics. We can introduce the same concept of "object class semantics" to matrices, by specifying an object class for columns and another for rows.

Definition 3.3 (Applied Matrix). An $AppliedMatrix$ is defined as a $Matrix$, the class of objects, which the individual column-vectors' components are based on, or refer to, and a class of object to describe the index semantics valid for all columns.

$$\lceil AppliedMatrix \rfloor_X = \lceil Matrix \rfloor_X \times \lceil ObjectClass \rfloor \times \lceil ObjectClass \rfloor \tag{155}$$

$$AppliedMatrix = (Matrix_X, ColumnClass, RowClass). \tag{156}$$

This means that a column vector can be expressed as an applied vector as follows.

$$AppliedVector = (column_{Matrix}(Matrix_X), ColumnClass). \tag{157}$$

Analogously to bound vectors, we can introduce the notion of a bound matrix.

Definition 3.4 (Bound Matrix). A bound matrix introduces an assignment individual objects to every column and every row. Denoted by $BoundMatrix$, it

is defined as follows.

$$\begin{aligned}(158)\lceil BoundMatrix\rfloor_X &= \lceil Matrix\rfloor_X \times (\lceil ObjectClass\rfloor \times \lceil List\rfloor_{\lceil Object\rfloor})^2\\ BoundMatrix_X &= (Matrix_X, ColumnClass,\\ &\quad ColumnObjects_{\lceil Object\rfloor}, RowClass,\\ &\quad RowObjects_{\lceil Object\rfloor}).\end{aligned}$$

We can use an analogous way of extracting column vectors as in Definition 3.3.

Let us now turn to the semantics of individual values.

3.3. Value Semantics. As we have defined that a value cannot have the type of an object reference, they can be used much like a vector with only a single dimension. This means that it may be useful to specify the semantics of a value analogously to that of vectors.

This consideration leads to the definitions of applied and bound values, as given below.

Definition 3.5 (Applied Values). An $AppliedValue$ is defined as a $Value$ and the class of objects, which this value is based on, or refers to.

$$\lceil AppliedValue\rfloor_X = \lceil Value\rfloor_X \times \lceil ObjectClass\rfloor \tag{159}$$

$$AppliedValue = (Value_X, ObjectClass). \tag{160}$$

Definition 3.6 (Bound Values). A $BoundValue$ is the extension to an applied value, defined analogously to applied and bound vectors. It also provides the object from the object class, which is involved in this value.

A $BoundValue$ is defined as follows.

$$\lceil BoundValue\rfloor_X = \lceil Value\rfloor_X \times \lceil ObjectClass\rfloor \times \lceil Object\rfloor \tag{161}$$

$$BoundValue_X = (Value_X, ObjectClass, ReferenceObject). \tag{162}$$

3.4. List Semantics. The main difference between a list and a vector is the fact that lists can contain actual objects (or references thereof). For our model, the most interesting case is that of a list containing *only* objects from *only one* object class. We will refer to such a list as an *object* list, denoted by $ObjectList$.

Definition 3.7 (Object Lists). An object list is a specialization of a general list, as an object list contains only entries of type $Object$, and all entries belong to a denoted class of objects. It is defined as follows:

$$\lceil ObjectList\rfloor = \lceil List\rfloor_{Object} \times \lceil ObjectClass\rfloor \tag{163}$$

$$ObjectList = (\langle List_{Object}\rangle, ObjectClass), \tag{164}$$

such that the following holds for $ObjectList$:

$$\forall 1 \leq i \leq n : \langle List_{Object}\rangle_i \in ObjectClass. \tag{165}$$

As we aim do develop a model suitable for artificial intelligence, we will also introduce a special form of the typed list, that of a *context type* list, which hosts context entities of a single, desinated type.

Definition 3.8 (Context Type Lists). A context type list is a list of abstract context entities, which all belong to the same context type X.

$$\lceil CTList \rfloor_X = \left\lceil \langle^n \lceil Context \rfloor \rangle \right\rfloor, \tag{166}$$

$$CTList_X = \langle CTList \rangle \tag{167}$$

such that the following holds for all $\langle CTList \rangle_i$:

$$\forall 1 \leq i \leq n : T(\langle CTList \rangle_i) = X \in \lceil ContextType \rfloor. \tag{168}$$

This concludes our extended list vocabulary.

3.5. Assignment Semantics. As we know, an assignment is a mapping from one object class to another. It is very difficult to define a way of specifying semantics for assignments in a similar way as for vectors or values. For our model, we will only consider those assignments, which are based on object classes. Assignments often serve as translations between object classes or for specifying attributes.

Now consider the case where we have a value for every user, which specifies a *single* document of interest for a user. It is mathematically sound to use a translation from *documents* to *web-sites* in order to obtain a web-site of interest for this user. The decision wheather or not this is any useful information, and the interpretation of the *concise* semantics of such a composition must be done by a human, but this way of obtaining *possible* new context types, based on the domain and codomain of an assignment, can be used to show possible extensions of a model.

If we are dealing with an applied vector, we assume that we can apply the assignment on the *semantics* of the vector. This means that if we have an applied vector *about* topics, and an assignment mapping topics to words, we can very easily obtain a vector *about words*.

If we are dealing with an object list, we can use a suitable assignment to translate the entire list, by applying the function component-wise to the list values. The only requirement is that the domain of the assignment matches the object class of the object list.

4. Type Concepts

As previously discussed, a type concept is a template for data-structures that allows some additional specification of semantics. In fact, we already have all definitions necessary to define a type concept.

Definition 4.1 (Type Concepts and Labeled Type Concepts). The set of all type concepts $\lceil TypeConcept \rfloor$ is the collection of all structure types and their sematically described variants.

$$\begin{aligned} \lceil TypeConcept \rfloor = \big\{ & T \,|\, T \in \lceil Value \rfloor \vee T \in \lceil Vector \rfloor \\ \vee \; & T \in \lceil Matrix \rfloor \vee T \in \lceil List \rfloor \\ \vee \; & T \in \lceil Assignment \rfloor \\ \vee \; & T \in \lceil AppliedValue \rfloor \vee T \in \lceil BoundValue \rfloor \\ \vee \; & T \in \lceil AppliedVector \rfloor \vee T \in \lceil BoundVector \rfloor \\ \vee \; & T \in \lceil AppliedMatrix \rfloor \vee T \in \lceil BoundMatrix \rfloor \\ \vee \; & T \in \lceil ObjectList \rfloor \vee T \in \lceil CTList \rfloor \big\}. \end{aligned} \tag{169}$$

For creating the specification of attributes in object oriented models, we only need to introduce suitable labels.

$$\lceil IDTypeConcept \rfloor = \lceil TypeConcept \rfloor \times \lceil Identifier \rfloor. \tag{170}$$

We will next consider how these considerations can be applied to context aware systems.

CHAPTER 4

Inferring Typed Context Types

1. Basic Rules for Inferring Types

In this chapter, we will now turn to the task of inferring context types with corresponding type concepts. As we recall, we use our initial context connectivity rule $\circ$ as an initial decision on what context types and groups are "basically composable". But this rule alone will not suffice for any advanced considerations on context types.

We will need additional rules, which express our considerations about several "simple default operations", allowing us to derive new context types with little development effort. We will devise a set of rules, allowing us to express the considerations on possible merging operations or similarity computation. These possible model extensions are of course subjected to the basic design principles in AI, meaning that even though they may be mathematically sound according to the model, and may be predicted, corrected, completed or reproduced by the underlying algorithm used for and on the representation, there is no *mathematical* way of knowing the correctness of the "real world interpretation and theory". We cannot attempt to solve this problem, but try to aid the developers in recognizing possible system extensions and algorithmic spin-offs to the system design.

When we view this process while considering the classical problem of frames (as in [**4**]) and the real world, we can view the system design support as we evision it as an attempt to automatically deduce new frames based on the description of the initial frames, as designed by the developer. This cannot solve the basic "frames vs. reality" problem the field of artificial intelligence faces, but evolves around the idea of simplifying the task of designing modern systems with AI methods.

As a first step towards inferred context types, we need to associate context types with type concepts. To keep this binding loose, but secure at the same time, we will wrap the process of typing in a function.

1.1. Typing Context Types. The initial decision of how to craft type concepts for context types lies with the developer. We will use a function template to represent the type concept assignments in the system, denoted by $\lceil \mu \rfloor$.

Definition 1.1 (Assigning Type Concepts to Context Types). For the purpose of determining which type concept is to be used for representing the data-type of a context type, we now introduce a function template, denoted by μ.

$$\lceil \mu \rfloor = \Big\lceil \lceil ContextType \rfloor \quad \rightarrow \quad \lceil TypeConcept \rfloor \Big\rfloor, \tag{171}$$

which contains the actual element-by-element assignment according to the system design.

1.2. Type Concept Connectivity. We will introduce a form of connectivity based on structure types with additional specification of semantics. As before with context groups, we will introduce a rule for the composability of type concepts. We will first show the complete definition of this rule, and then discuss its meaning in parts.

Definition 1.2 (Connectivity Rule for Type Concepts). We will now define a connectivity rule $\bowtie$ for type concepts.

(172) $A \bowtie B$ *holds, iff*

$$A = (V, C) \in \lceil AppliedVector \rfloor \wedge \tag{173}$$
$$\wedge B = (V', C') \in \lceil AppliedVector \rfloor \wedge$$
$$\wedge dim(V) = dim(V') \wedge C = C'$$
$$\vee$$
$$A = (V, C, R) \in \lceil BoundVector \rfloor \wedge \tag{174}$$
$$\wedge B = (V', C', R') \in \lceil BoundVector \rfloor \wedge$$
$$\wedge dim(V) = dim(V') \wedge C = C' \wedge R = R'$$
$$\vee$$

$$A = (E, C) \in \lceil AppliedValue \rfloor \wedge \tag{175}$$
$$\wedge B = (E', C') \in \lceil AppliedValue \rfloor \wedge$$
$$\wedge C = C'$$
$$\vee$$
$$A = (E, C, R) \in \lceil BoundValue \rfloor \wedge \tag{176}$$
$$\wedge B = (E', C', R') \in \lceil BoundValue \rfloor \wedge$$
$$\wedge C = C' \wedge R = R'.$$
$$\vee$$

$$A = (V, C) \in \lceil AppliedVector \rfloor \wedge \tag{177}$$
$$\wedge B \in \lceil Assignment \rfloor \wedge$$
$$\wedge dom(B) = C$$
$$\vee$$
$$A = (V, C, R) \in \lceil BoundVector \rfloor \wedge \tag{178}$$
$$\wedge B \in \lceil Assignment \rfloor \wedge$$
$$\wedge dom(B) = C$$
$$\vee$$

$$A = (E, C) \in \lceil AppliedValue \rfloor \wedge \wedge B \in \lceil Assignment \rfloor \wedge \wedge dom(B) = C \tag{179}$$

$$\vee$$

$$A = (E, C, R) \in \lceil BoundValue \rfloor \wedge \wedge B \in \lceil Assignment \rfloor \wedge \wedge dom(B) = C \tag{180}$$

$$\vee$$

$$A \in \lceil Assignment \rfloor \wedge \wedge B \in \lceil Assignment \rfloor \wedge \wedge codom(A) = dom(B) \tag{181}$$

$$\vee$$

$$A = (L, C) \in \lceil ObjectList \rfloor \wedge \wedge B \in \lceil Assignment \rfloor \wedge \wedge dom(B) = C \tag{182}$$

$$\vee$$

$$A \in \lceil CTList \rfloor_X \wedge \wedge B \in \lceil CTList \rfloor_{X'} \wedge \wedge length(A) = length(B) \wedge \mu(X) \bowtie \mu(X'), \tag{183}$$

where μ is a member of the function template μ for accessing the type concept of a given context type.

Let us now consider the individual parts of this definition.

Vector: As in Statements 173 and 174, we consider those vectors composable, for which the index semantics are specified over the same object class. A typical composition function could be vector similarity, which results in a single, numerical value. Bound vectors must furthermore have matching objects assigned to the individual indices.

Value: As in Statements 175 and 176, we consider values composable if they have a matching object classes in their semantic descriptors, and a matching object for bound values.

Assignment: As in Statements 177, 178, 179, and 180, we define that assignments can be used to translate index semantics. This assumption is based on the fact, that we can use the same assignment to translate the objects from the corresponding object classes.
Like functions, we can also compose assignments, if the image of the first function is an assignment to the preimage of the second function, as expressed in Statement 181.
Furthermore, as in Statement 182, we can also use assignments to translate object lists, by applying it to every entry of the list.

CTList: As in Statement 183, we can basically combine these lists of context types, if they have an equal length and entries of context types with composable type concepts.

$\circ$	$\bowtie$	$\Diamond$	$\triangle\, \nabla\, \lhd\, \rhd$
Context Connectivity	Type Concept Connectivity	Full Connectivity	Further Composite Rules

FIGURE 1. Notation and Application

We will now turn to the rules we can derive from $\circ$ and $\bowtie$.

1.3. Composite Rules. For modeling purposes, it is of advantage to have a small vocabulary of composite rules (see Figure 1). These rules are composed from our two basic rules, $\circ$ and $\bowtie$, and μ typing functions.

DEFINITION 1.3 (Composite Rules). For modeling support we define several rules over relations. The $\Diamond$ rule expresses full composability, both in terms of context type and type concept connectivity.

$$\begin{aligned} \Diamond &\subseteq \lceil ContextType \rfloor \times \lceil ContextType \rfloor \\ A \Diamond B &\Leftrightarrow A \circ B \wedge \mu(A) \bowtie \mu(B), \end{aligned} \tag{184}$$

for a given μ.

The $\triangle$ rule can be used for type checking.

$$\begin{aligned} \triangle &\subseteq \lceil ContextType \rfloor \times \lceil TypeConcept \rfloor \\ A \triangle T &\Leftrightarrow \mu(A) = T, \end{aligned} \tag{185}$$

for a given μ.

The ∇ rule can be used to check for context types, which are underspecified because they have not been assigned a type concept in a μ function.

$$\begin{aligned} \nabla &\subseteq \lceil ContextType \rfloor \times \lceil \mu \rfloor \\ A \nabla m &\Leftrightarrow A \ni image(m). \end{aligned} \tag{186}$$

The $\rhd$ rule expresses that two context types have connectivity based on $\circ$, but the assigned type concepts prohibit automated support, as the $\bowtie$ rule does not hold.

$$\begin{aligned} \rhd &\subseteq \lceil ContextType \rfloor \times \lceil ContextType \rfloor \\ A \rhd B &\Leftrightarrow A \circ B \wedge \neg(\mu(A) \bowtie \mu(B)), \end{aligned} \tag{187}$$

for a given μ.

The $\lhd$ rule expresses that two context types are completely unconnected, as neither $\circ$ nor $\bowtie$ hold.

$$\begin{aligned} \lhd &\subseteq \lceil ContextType \rfloor \times \lceil ContextType \rfloor \\ A \lhd B &\Leftrightarrow \neg(A \circ B) \wedge \neg(\mu(A) \bowtie \mu(B)), \end{aligned} \tag{188}$$

for a given μ.

We will now see how these rules can allow us to automatically deduce new context types with type concepts.

	Value	Vector	Assignment	Object List	Context Type List
Value	d_R, m_R	$\times'$	t_{sem}	–	*sub*
Vector	-	d_{R^n}	t_{sem}	–	*sub*
Assignment	-	-	$\circ'$, CSS	t_{obj}	*sub*
Object List	-	-	-	–	*sub*
Context Type List	-	-	-	-	*sub*

TABLE 1. Types and Simple Methods - Overview

2. Methods and Type Concepts

The task of deducing new context types can be split into several tasks.

- Deducing the method.
- Deducing the type concept.
- Deducing object class and view object class of the new context type.

These tasks are an actual deduction *inside* our descriptive framework, but our assumptions on method level, which also have an impact on the type concept, and the general nature of our rules yield a result which is a recommendation to a system developer rather than a "rock solid" deduction.

We will now turn to the first task, the deduction of methods, and study their implications on type concepts.

2.1. Inferring Methods. If we consider our basic structure types, namely values, vectors, matrices, lists, assignments and their semantically extended variants, we can recommend a few very basic methods, which can be used to combine context types in some manner, and are very commonly used in AI for certain types and encodings. This means that the recommendation is mathematically sound for the respective structure type, and mostly based on commonly agreed methods in AI, but there is no general mathematical way of knowing the overall soundness of the entire deduced context type.

If we are dealing with semantically specified type concepts, we can use methods as depicted in Table 1 and Table 2, which use short names for the type concepts for readability.

We will now discuss the application of these methods to our sematically described type concepts in detail.

d_R - If we consider the application of value distance to applied and bound values, we should discriminate between two cases:

Number: We are comparing two values, which are both about the same object class. In the most simple case, this could be merely a comparison of the amount of water in a glass.

Other: These similarities can span a broad range of comparisons, ranging from a rather simple comparison of boolean values using $\wedge$, $\vee$ or any other binary operator, to complex string comparisons. The basic

Symbol	Function Type	Example Function
d_R	single value distance	$d_R(a,b) = \|a-b\|$
m_R	combined value metric	$m_R(a,b) = \frac{a+b}{2}$
d_{R^n}	vector similarity	$d_{R^n}(a,b) = c\vec{o}s(a,b)$
$\times'$	scalar multiplication	$\times'(v,k) = kv$
$\circ'$	function composition	$(g \circ' f)(x) = g(f(x))$
CSS	context similarity substitution	-
t_{sem}	semantic object class translation	$f : A \to B$ $a \mapsto b$
t_{obj}	object class translation	$f : A \to B$ $a \mapsto b$
sub	compute from nested context type	-
$-$	no composition possible	-

TABLE 2. Types and Simple Methods - Denotation

principle of matching object class semantics and, for bound values, object must hold (as per $\bowtie$).

m_R - A combined metric can be used to craft new rating schemes from several existing context types.

d_{R^n} - Basic vector similarity can, according to our definitions, be applied to vectors with equal dimensionality and matching index semantics. This is motivated by the fact that, in standard vector similarity, we can compare all vectors with *term* index semantics, regardless of whether we are dealing with a document representation, an artificially crafted class representation or user interest representation vector. Bound vectors also include the "dictionary" by hosting the individual term for each index.

Non-numerical vectors can also be compared, if they are translated into numerical representation prior to the computation of similarity.

$\times'$ - Scalar multiplication, a key requirement for any vector space, can also be applied, but the requirements can not be expressed in the $\bowtie$ rule alone. The basic principle for scalar multiplication context type composition is that it is allowed, if the *semantics* of the value match the "source" object class of the second context type.

This means that if we have a value for describing a user's interest in a document, and a vector representation of the document - topic dependencies, we can obtain a re-scaled topic vector for the user.

$\circ'$ - Function composition can be applied to assignments, given the common requirement.

$$(189) \qquad (g \circ' f)(x) = g(f(x)) \ \ldots \ \textit{iff } image(f) = preimage(g).$$

Database theory would suggest considerations about cardinality, but as we do not have to resort to inversions, these are not necessary here.

CSS - This operation corresponds to the combination of two parallel translations with assigments by applying one or more other context types to compare the results of the translations.

Symbol	Function Type	Result Structure Type
d_R	single value distance	$Value$
m_R	combined value metric	$Value$
d_{R^n}	vector similarity	$Value$
$\times'$	scalar multiplication	$Vector$
$\circ'$	function composition	$Assignment$
t_{sem}	semantic object class translation	$id_{StructureType}$
t_{obj}	object class translation	$id_{StructureType}$
sub	compute from nested context type	$List$

TABLE 3. Simple Methods and Result Structure Type

t_{sem} - The translation of object class semantics seems a bit problematic at first sight, but the principle becomes apparent if you consider expansions by more than one step. The consideration to translate semantics is based on the fact that we can not only translate from the original object class semantics to the new object class for semantic description, but also between the corresponding object classes in *exactly* the same manner. This means that we can translate all object involved, especially those in bound values or vectors, in the same manner, thus making the operation possible in running systems.

t_{obj} - The translation of objects in object lists using assignments is a classical operation.

sub - With context type list, we essentially have to check the connectivity rule $\bowtie$ for the type concept of the list's "embedded" context type, but have to substitute the inferred context type back into a context type list.

We will now consider the "return types" of these methods.

2.2. Inferring Type Concepts. If we recapture our set of methods in Table 2, we can make some basic assumptions about the resulting structure types. These considerations are depicted in Table 3, where $id_{StructureType}$ stands for the identity function on structure types.

If we now turn to semantics, we have a rather simple situation if assignments are used for translations. These can now also be used to translate the objects in object lists, index objects of bound vectors and the assigned object of a bound value. Regarding the semantics of value and vector similarity, the situation is a little more complex.

d_{R^n}, d_R - It is hard to specify the semantics of vector or value similarity beyod the point of specifying that the result, in fact, constitutes some kind of similarity. This similarity again forms a context type, but the *meaning* of this context type can only be deduced from the two underlying context types used to create the new type using similarity. There are several ways, in which we can handle this situation.

(i) We use a semantically unspecified type concept for the new context type.

(ii) We use an object class *Similarity* to describe the semantics.
(iii) We use the *view object class* of the resulting new context type to describe the semantics of the result, and use the semantically described variants of the structure types in Table 3.
(iv) We retain the original object class semantics.

We will choose option (iv), as we assume that a new rating is deduced *over* these object class semantics, meaning that if we combine the following context types:

$$(TopicContext, Documents, Topics) \quad \triangle \quad TopicVector \tag{190}$$
$$\Diamond$$
$$(UserInterest, Users, Topics) \quad \triangle \quad TopicVector, \tag{191}$$
$$TopicVector \quad = \quad (v, Topics), \tag{192}$$

we obtain a new context type:

$$(VectorSimilarity, Users, Documents) \quad \triangle \quad (val, Topics). \tag{193}$$

m_R - We can craft a combined rating from several contexts of the form

$$(m_i, A_i, B_i) \quad \triangle \quad X, \tag{194}$$
$$X \quad \in \quad \{AppliedValue, BoundValue\}, \tag{195}$$

leading to a combined rating, both in terms of value and semantics.

If we have two context types for user similarity, over both interest and spatial distance, namely

$$(InterestSimilarity, User, User) \quad \triangle \quad (val, Interest) \tag{196}$$
$$\circ$$
$$(SpatialAffinity, User, User) \quad \triangle \quad (val, Positions), \tag{197}$$

we can construct the following combined metric from these two typed context types:

$$(CombinedMetric, User, User) \quad \triangle \quad (val, User). \tag{198}$$

$\times'$ - If we consider the scalar multiplication, we notice that it does not change any index semantics in the vector. If we have a value for a user's interest in a document, and a topic vector for every document, and we re-scale the topic vector over the user interest, the index semantics are still described using an object class for topics.

Let us examine this example more closely:

$$(UserDocumentInterest, Users, Documents) \quad \triangle \quad (val, Documents) \tag{199}$$
$$\triangleright$$
$$(TopicContext, Documents, Topics) \quad \triangle \quad (vec, Topics). \tag{200}$$

thus yields

$$(ScalarMultiplication, Users, Topics) \quad \triangle \quad (vec, Topics). \tag{201}$$

As we see, even though the full composition rule $\Diamond$ does not hold, we can still find a composition.

$\circ'$ - Assignment composition again follows the rules for ordinary function composition $\circ'$. This means that for two assignments,

$$(F2E, FrenchWords, EnglishWords) \quad \triangle \quad A \in Assignment \tag{202}$$
$$dom(A) \quad = \quad FrenchWords \tag{203}$$
$$codom(A) \quad = \quad EnglishWords \tag{204}$$
$$\Diamond$$
$$(E2Topics, EnglishWords, Topics) \quad \triangle \quad B \in Assignment \tag{205}$$
$$dom(B) \quad = \quad EnglishWords \tag{206}$$
$$codom(B) \quad = \quad Topics, \tag{207}$$

we can obtain a new assignment

$$(Composition, FrenchWords, Topics) \quad \triangle \quad C \in Assignment, \tag{208}$$
$$dom(C) \quad = \quad dom(A) \tag{209}$$
$$codom(C) \quad = \quad codom(B). \tag{210}$$

CSS - Context similarity substitution combines the results of two parallel translation using a third context. This means that we can combine two context types of the form:

$$(m, A, B)\triangle Assignment \quad \wedge \quad (m', C, D)\triangle Assignment', \tag{211}$$
$$image(Assignment) \quad = \quad X \tag{212}$$
$$image(Assignment') \quad = \quad Y, \tag{213}$$

if we have additional context types of the form:

$$(m_{XY}, X, Y) \quad \triangle \quad T_{XY} \tag{214}$$
$$(m_X, X, X) \quad \triangle \quad T_X \tag{215}$$
$$(m_Y, Y, Y) \quad \triangle \quad T_Y, \tag{216}$$

to a new context type:

$$(CSS, A, C)\triangle T_{AC}, \tag{217}$$

where T_{AC} is the resulting type of a deduction from T_{XY}, T_X and T_Y, as depicted in Figure 2.

t_{sem} - The translation of semantics is more complicated in terms of type mapping than in the basic concept. For comprehensability, let us assume that we use a function template S for mapping the semantic objects, and mapping the semantic object classes from the domain of the assignment to its codomain. This means that a function template S with the following structure:

$$S : \lceil TypeConcept \rfloor \quad \rightarrow \quad \lceil TypeConcept \rfloor, \tag{218}$$

is contained in the implementation of the assignment.

t_{obj} - The translation of objects in an object list, or other objects, is again represented by the S function template, which represents the process of applying the function inside the assignment to every element of the list, and replacing the object class of the type specification with the codomain of the assignment.

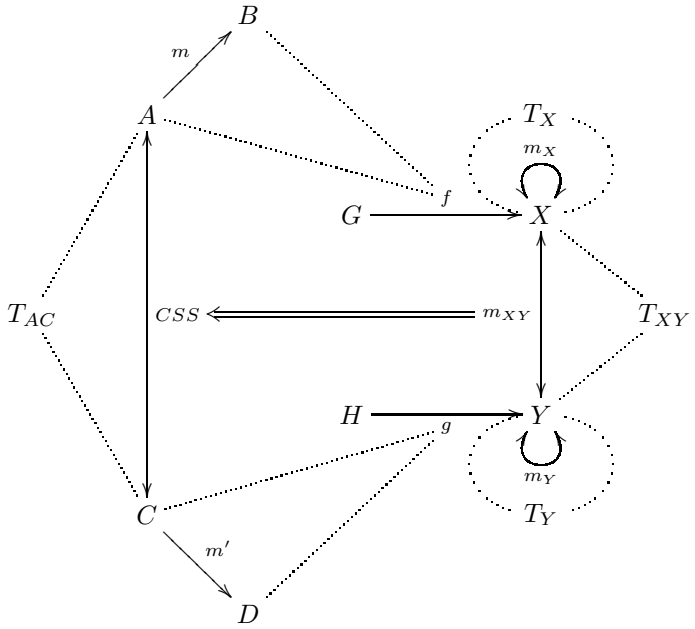

FIGURE 2. Context Similarity Substitution Sketch

Symbol	Deductions
d_R	$AppliedValue$
m_R	$AppliedValue$
d_{R^n}	$AppliedValue$
$\times'$	$id_{AppliedVector} \vee id_{BoundVector}$
$\circ'$	$Assignment$
t_{sem}	S
t_{obj}	S_{obj}
sub	replace individually inferred context type

TABLE 4. Methods and Type Concepts

sub - For processing context types in a context type list, the *sub* function applies the type inferencing process on the nested context type and the corresponding type concept, and substitutes the new context type in the context type list.

Considering our options, we can subsume our type concept deductions in Table 4.

3. Unary List Operations

We now have several operations for combining two context types, but lists allow several operations without any additional context types. These operations are unary list operations.

We will first define a simple operation responsible for extracting elements from context type lists.

Definition 3.1 (Object List Element Selection). For a given object list and a selection predicate p_s, we can obtain the first entry from the list, for which p_s holds.

$$\begin{aligned} select : \lceil ObjectList \rfloor_X \times \lceil p_s \rfloor &\rightarrow \lceil X \rfloor \\ (list, type, p_s) &\mapsto l_i \in list \text{ } iff \text{ } p_s(l_i). \end{aligned} \tag{219}$$

We can also select parts of an object list in a similar manner.

Definition 3.2 (Typed Sub-List Selection). For a given object list and a selection predicate p_s, we can obtain all entries from the list, for which p_s holds.

$$\begin{aligned} sub : \lceil ObjectList \rfloor_X \times \lceil p_s \rfloor &\rightarrow \lceil X \rfloor \\ (list, type, p_s) &\mapsto \{l | p_s(l)\}. \end{aligned} \tag{220}$$

We also have the notion of a context type list. Direct modeling of context type lists is not very frequently used, but there is an operation which can turn an object list into a context type list.

Definition 3.3 (Context Type Substitution in Object Lists). If we are dealing with an object list, we can substitute the object entries by matching context entities. On type level, this means the following:

$$(list, A) \wedge (method, A, B) \rightsquigarrow (list, (method, A, B)) \in \lceil CTList \rfloor. \tag{221}$$

We will now turn to the operations on context type lists.

3.1. Context Type List Operations. We now define a selection function for context type lists.

Definition 3.4 (Context Type List Element and Sub-List Selection). For a context type list and a selection predicate p_s, we can obtain the first entry from the list, for which p_s holds.

$$\begin{aligned} select : \lceil CTList \rfloor \times \lceil p_s \rfloor &\rightarrow \lceil Context \rfloor \\ (\langle L \rangle, t, p_s) &\mapsto \langle L \rangle_i \text{ } iff \text{ } p_s(\langle L \rangle_i). \end{aligned} \tag{222}$$

We also have a function for accessing a sub-list.

$$\begin{aligned} sub : \lceil CTList \rfloor \times \lceil p_s \rfloor &\rightarrow \lceil Context \rfloor \\ (\langle L \rangle, t, p_s) &\mapsto \left\{ l_i \middle| p_s(\langle L \rangle_i) \right\}. \end{aligned} \tag{223}$$

Using this function, we can now create new context types, based on the selection of a list element in a context type list. If we have a context type, which is typed to a context type list, mathematically:

$$(m, A, B) \triangleq (list, (m', A', B')), \tag{224}$$

and a selection predicate p_s, we can derive a new context type, namely:

$$(p_s, A', B') \triangleq \mu((m', A', B')). \tag{225}$$

We can use the *sub* function analogously, which represents a query mechanism. For a context type with type concept

$$(m, A, B) \triangleq (list, (m', A', B')), \tag{226}$$

and a selection predicate p_s, we can derive a sub-list context type, namely:

$$(p_s, A, B) \triangleq (list, (m', A', B')). \tag{227}$$

With suitable selection predicates, these methods of deducing new context types lead to the ability to make recommendations in the system.

Another operation on context type lists is the replacement of the entities of the context type by the actual values of the context entities. This leads to an operation, which turns a context type list into an object list, in which all entries are of the same structure type as the structure type of the original context type.

This means that we can turn the following context type list:

$$(ListContext, A, B) \quad \triangle \quad (ctlist, (m, X, Y) \triangle T), \tag{228}$$

$$T \quad \in \quad \lceil TypeConcept \rfloor, \tag{229}$$

into the following object list:

$$(ListContext^{\star}, A, \tau) \quad \triangle \quad T, \tag{230}$$

$$T = (val, X) \quad \Rightarrow \quad \tau = X, \tag{231}$$

$$T = (val, X, O) \quad \Rightarrow \quad \tau = X, \tag{232}$$

$$T = (vec, X) \quad \Rightarrow \quad \tau = X, \tag{233}$$

$$T = (vec, X, O) \quad \Rightarrow \quad \tau = X, \tag{234}$$

$$T = (mat, C, R) \quad \Rightarrow \quad \tau = C, \tag{235}$$

$$T = (mat, C, O_C, R, O_R) \quad \Rightarrow \quad \tau = C, \tag{236}$$

$$T = f \quad \Rightarrow \quad \tau = image(f), \tag{237}$$

$$T = (list, T') \quad \Rightarrow \quad \tau = T'. \tag{238}$$

3.2. System Context Types. For all objects known to the system, we need to have an initial means of directly accessing these. We will model this using system context types. A system context type is simply an object list for one of the object classes used in a system. By constructing, and maintaining, a list for every object class, we can now access all objects, if the object classes are a complete partition of the objects.

We will now introduce a function $\exists_{Sys}$ for generating such system context lists.

Definition 3.5 (System Exist Context). We can obtain a system exist context for an object class using the function $\exists_{Sys}$.

$$\exists_{Sys} : \lceil ObjectClass \rfloor \quad \rightarrow \quad \lceil ContextType \rfloor \tag{239}$$

$$C \quad \mapsto \quad (SystemExist, System, C),$$

such that the following holds:

$$(SystemExist, System, C) \quad \triangle \quad (list, C), \tag{240}$$

$$(list, C) \quad \in \quad \lceil ObjectList \rfloor \tag{241}$$

$$System \quad = \quad \{\sigma\}, \tag{242}$$

and σ is unique.

As the system exist context type is our direct access to objects, it is also a convenient way of knowing the queries possible to a system, namely all context types, which can be traced to a system exist context.

4. Type Change Methods

Several structure types have implicit relationships. Vectors can be composited from values, just as matrices can be composed from vectors. We now aim to study the deductions possible over these implicit relationships, in conjunction with our model for specifying semantics.

4.1. Vector and Matrix Construction. As vectors are a very convenient representation, we will now use the semantics of the system exist context to create vector representations. We will need no further way of specifying "existence semantics" - the system exist context types are uniquely defined over the unique object class $System$[1] and the designated $SystemExist$ method, we have a direct means of discerning system context types and ordinary context types.

The construction principle is very simple:

$$(SystemExist, System, B) \wedge (m, A, B) \triangle AppliedValue \tag{243}$$
$$\rightsquigarrow$$
$$(m, A, B) \triangle AppliedVector. \tag{244}$$

We can now assume the following about $AppliedVector$:

$$dim(AppliedVector) = length(ObjectList) \text{ iff} \tag{245}$$
$$(SystemExist, System, B) \triangle ObjectList,$$
$$AppliedValue = (val, C) \wedge AppliedVector = (vec, C') \Rightarrow \tag{246}$$
$$C = C'.$$

We can proceed analogously for bound vectors. Analogously, we can also compose matrices, if we have a vector as type concept of the underlying context type.

$$(SystemExist, System, B) \wedge (m, A, B) \triangle AppliedVector \tag{247}$$
$$\rightsquigarrow$$
$$(m, A, B) \triangle AppliedMatrix, \tag{248}$$

such that the following holds for $AppliedMatrix$:

$$(m, A, B) \triangle (vec, C) \wedge AppliedMatrix = (mat, B', C') \Rightarrow \tag{249}$$
$$B = B' \wedge C = C'.$$

Again, we can proceed analogously for bound matrices.

4.2. Selection from Vectors and Matrices. As an analogous operation to selection from lists, we can also use selection predicates to extract vectors from matrices and values from vectors, leading to new typed context types. Let us first define the elementary functions for type concepts.

Definition 4.1 (Matrix Column Selection). We define a function $select_{col}$ for selecting a single column from an applied matrix.

$$select_{col} : \lceil AppliedMatrix \rfloor_X \times \lceil p_s \rfloor \rightarrow \lceil AppliedVector \rfloor_X \tag{250}$$
$$\big((mat, C, R), p_s\big) \mapsto (vec, R'),$$

[1] As σ is unique, $System$ is a unique set with only one element.

such that the following holds for (vec, R'):

$$p_s(vec) \quad \wedge \quad R = R', \tag{251}$$

and vec is a column of mat:

$$\exists i, 1 \leq i \leq m \quad : \quad vec = column_{matrix}(mat, i). \tag{252}$$

We have an analogous function for bound matrices.

$$\begin{aligned} select_{col} : \lceil BoundMatrix \rfloor_X \times \lceil p_s \rfloor \quad &\rightarrow \quad \lceil AppliedVector \rfloor_X \\ \big((mat, C, o_c, R, o_r), p_s\big) \quad &\mapsto \quad (vec, R', o'_r), \end{aligned} \tag{253}$$

such that the following holds for (vec, R', o'_r):

$$p_s(vec) \wedge R = R' \wedge o_r = o'_r. \tag{254}$$

and vec is again a column of the matrix mat.

Definition 4.2 (Matrix Row Selection). We also define a function $select_{row}$ for selecting a single row from an applied matrix.

$$\begin{aligned} select_{row} : \lceil AppliedMatrix \rfloor_X \times \lceil p_s \rfloor \quad &\rightarrow \quad \lceil AppliedVector \rfloor_X \\ \big((mat, C, R), p_s\big) \quad &\mapsto \quad (vec, C'), \end{aligned} \tag{255}$$

such that the following holds for (vec, C'):

$$p_s(vec) \quad \wedge \quad C = C', \tag{256}$$

and vec is a row of mat:

$$\exists i, 1 \leq i \leq m \quad : \quad vec = row_{matrix}(mat, i). \tag{257}$$

Analogously, we have a row selection function for bound matrices.

$$\begin{aligned} select_{row} : \lceil BoundMatrix \rfloor_X \times \lceil p_s \rfloor \quad &\rightarrow \quad \lceil AppliedVector \rfloor_X \\ \big((mat, C, o_c, R, o_r), p_s\big) \quad &\mapsto \quad (vec, C', o'_c), \end{aligned} \tag{258}$$

such that the following holds for (vec, C', o'_c):

$$p_s(vec) \wedge C = C' \wedge o_c = o'_c. \tag{259}$$

and vec is again a row of the matrix mat.

We can now deduce new typed context types as follows:

$$\begin{aligned} (m, A, B) \triangle (mat, C, R) \quad &\rightsquigarrow \quad (select_{col}, A, B) \triangle (vec, R), \\ (vec, R) \quad &= \quad select_{col}(mat, C, R), \end{aligned} \tag{260}$$

$$\begin{aligned} (m, A, B) \triangle (mat, C, R) \quad &\rightsquigarrow \quad (select_{row}, A, R) \triangle (vec, C), \\ (vec, C) \quad &= \quad select_{row}(mat, C, R). \end{aligned} \tag{261}$$

We can proceed analogously for bound matrices. Regarding the selection of values from vectors, we again define a selection function:

Definition 4.3 (Vector Component Selection). We have a function $select_{comp}$ for selecting an individual component from a vector, based on a selection predicate p_s.

$$\begin{aligned} select_{comp} : \lceil AppliedVector \rfloor \times \lceil p_s \rfloor \quad &\rightarrow \quad \lceil AppliedValue \rfloor \\ (\langle vec \rangle, S) \quad &\mapsto \quad (val, S), \end{aligned} \tag{262}$$

so that $S(\langle vec \rangle_i)$ holds, and $val = \langle vec \rangle_i$.

We have an analogous function for bound vectors.

$$\begin{aligned} select_{comp} : \lceil BoundVector \rfloor \times \lceil p_s \rfloor &\rightarrow \lceil AppliedValue \rfloor \\ (\langle vec \rangle, S, O) &\mapsto (val, S, O'), \end{aligned} \tag{263}$$

so that $S(\langle vec \rangle_i)$ holds, $val = \langle vec \rangle_i$ and the following holds for O':

$$O' = o_i \in O \Leftrightarrow p_s(vec_i). \tag{264}$$

We can now extract a single element from a vector.

$$\begin{aligned} (m, A, B) \triangle (vec, S) &\rightsquigarrow (select_{comp}, A, S) \triangle (val, S), \\ (val, S) &= select_{comp}(vec, S). \end{aligned} \tag{265}$$

4.3. Context Type List Construction from Vectors. We can construct a context type list of applied or bound values from an applied or bound vector respectively. The deduction is as follows:

$$(m, A, B) \triangle (vec, S) \rightsquigarrow (build_{list}, A, B) \triangle \big((m, A, S) \triangle (val, S)\big), \tag{266}$$

$$(val, S) = select_{comp}((vec, S), p_s), \tag{267}$$

where p_s is a one-by-one selection predicate. This step back to context object lists concludes our type change methods.

Let us now subsume our considerations.

5. Context Type Deduction

In the previous sections of this chapter, we have studied the aspects of context type deduction on methods, structure types and semantically described type concepts. We will now summarize our considerations and present a set of deduction options. For compatibility with our general use of "arrow notation", we will use a notation of the form

$$\textit{premises} \rightsquigarrow \textit{possible deduction.} \tag{268}$$

We will first examine the deductions allowed due to the $\Diamond$ rule, and then study the list related deductions.

5.1. Basic Deductions. We can now summarize our possible deductions reflected in the $\bowtie$ rule. Note that we have chosen a single way of dealing with value and vector similarity semantics (see Section 2.2).

5.1.1. *Value Similarity.*

$$\begin{aligned} (m, A, B) \triangle AppliedValue \;\wedge\; (m', B, C) \triangle AppliedValue' \wedge \\ \wedge\, AppliedValue \bowtie AppliedValue' \\ \rightsquigarrow \\ (d_R, A, C) \triangle AppliedValue^\star \\ AppliedValue = (val, S) \Rightarrow AppliedValue^\star = (val^\star, S). \end{aligned} \tag{269}$$

5.1.2. *Combined Value Metric.*

$$\begin{aligned} (m, A, B) \triangle AppliedValue \;\wedge\; (m', A, B) \triangle AppliedValue' \\ \rightsquigarrow \\ (CombinedMetric, A, B) \triangle (val, B). \end{aligned} \tag{270}$$

5.1.3. *Scalar Multiplication.*

$$
\begin{array}{rcl}
(m,A,B)\triangle AppliedValue & \wedge & (m',B,C)\triangle AppliedVector \vee \\
\vee(m,A,B)\triangle AppliedVector & \wedge & (m',B,C)\triangle AppliedValue \wedge \\
\wedge\, AppliedValue & \bowtie & AppliedVector \\
& \rightsquigarrow & \\
(\times',A,C) & \triangle & AppliedVector^{\star} \\
AppliedVector = (vec,S) & \Rightarrow & AppliedVector^{\star} = (vec^{\star},S).
\end{array} \tag{271}
$$

5.1.4. *Vector Similarity.*

$$
\begin{array}{rcl}
(m,A,B)\triangle AppliedVector & \wedge & (m',B,C)\triangle AppliedVector' \wedge \\
\wedge\, AppliedVector & \bowtie & AppliedVector' \\
& \rightsquigarrow & \\
(d_{R^n},A,C) & \triangle & AppliedValue^{\star} \\
AppliedVector = (vec,S) & \Rightarrow & AppliedValue^{\star} = (val,S).
\end{array} \tag{272}
$$

5.1.5. *Assignment Composition.*

$$
\begin{array}{rcl}
(m,A,B)\triangle Assignment & \wedge & (m',B,C)\triangle Assignment' \wedge \\
\wedge\, image(Assignment) & = & preimage(Assignment') \\
& \rightsquigarrow & \\
(FunctionComposition,A,C) & \triangle & Assignment' \circ' Assignment.
\end{array} \tag{273}
$$

5.1.6. *Context Similarity Substitution.*

$$
\begin{array}{rcl}
(m,A,B)\triangle Assignment & \wedge & (m',C,D)\triangle Assignment' \wedge \\
\wedge\,(m_X,X,X)\triangle T_X & \wedge & (m_Y,Y,Y)\triangle T_Y \wedge \\
\wedge\,(m_{XY},X,Y) & \triangle & T_{XY} \wedge \\
\wedge\, image(Assignment) = X & \wedge & image(Assignment') = Y \\
& \rightsquigarrow & \\
(CSS,A,C) & \triangle & T_{AC} \\
& iff & \\
(m_X,X,X)\triangle T_X & \wedge & (m_{XY},X,Y)\triangle T_{XY} \wedge \\
\wedge\,(m_Y,Y,Y) & \triangle & T_Y \\
& \rightsquigarrow & \\
ContextType & \triangle & T_{AC}.
\end{array} \tag{274}
$$

5.1.7. *Object Class Semantics Translation.*

$$
\begin{array}{rcl}
(m,A,B)\triangle Assignment & \wedge & (m',B,C)\triangle Assignment' \\
\wedge\, Assignment & \bowtie & Assignment' \\
& \rightsquigarrow & \\
(\circ',A,C) & \triangle & Assignment^{\star} \\
Assignment^{\star} & = & Assignment \circ' Assignment'.
\end{array} \tag{275}
$$

$$
\begin{array}{rcl}
(m, A, B) \triangle AppliedValue & \wedge & (m', B, C) \triangle Assignment \vee \\
\wedge\, AppliedValue & \bowtie & Assignment \\
& \rightsquigarrow & \\
(t_{sem}, A, C) & \triangle & AppliedValue^{\star} \\
AppliedValue = (val, S) & \wedge & dom(Assignment) = S \wedge \\
\wedge\, codom(Assignment) = S' & \Rightarrow & AppliedValue^{\star} = (val^{\star}, S').
\end{array} \tag{276}
$$

$$
\begin{array}{rcl}
(m, A, B) \triangle AppliedVector & \wedge & (m', B, C) \triangle Assignment \vee \\
\wedge\, AppliedVector & \bowtie & Assignment \\
& \rightsquigarrow & \\
(t_{sem}, A, C) & \triangle & AppliedVector^{\star} \\
AppliedVector = (vec, S) & \wedge & dom(Assignment) = S \wedge \\
\wedge\, codom(Assignment) = S' & \Rightarrow & AppliedVector^{\star} = (vec^{\star}, S').
\end{array} \tag{277}
$$

5.1.8. *Object List Translation.*

$$
\begin{array}{rcl}
(m, A, B) \triangle ObjectList & \wedge & (m', B, C) \triangle Assignment \vee \\
\wedge\, ObjectList & \bowtie & Assignment \\
& \rightsquigarrow & \\
(t_{obj}, A, C) & \triangle & ObjectList^{\star} \\
ObjectList = (list, T) & \wedge & dom(Assignment) = T \wedge \\
\wedge\, codom(Assignment) = S & \Rightarrow & ObjectList^{\star} = (list^{\star}, S).
\end{array} \tag{278}
$$

5.2. Advanced Deductions. We will now examine those context type compositions, which go beyond the $\Diamond$ rule.

5.2.1. *Context Type Substitution.*

$$
\begin{array}{rcl}
(m, A, B) \triangle (list, X) & \wedge & (m', X, C) \\
& \rightsquigarrow & \\
(subst_{ContextType}, A, C) & \triangle & CTList^{\star} \\
CTList^{\star} & = & (list^{\star}, (m', X, C).
\end{array} \tag{279}
$$

5.2.2. *Context Entity Value Substitution.*

$$\begin{aligned}(280)\qquad (ListContext, A, B) &\triangle (ctlist, (m, X, Y)\triangle T) \wedge\\ T &\in \lceil TypeConcept\rfloor\\ &\rightsquigarrow\\ (ListContext^\star, A, \tau) &\triangle T,\\ T = (val, X) &\Rightarrow \tau = X,\\ T = (val, X, O) &\Rightarrow \tau = X,\\ T = (vec, X) &\Rightarrow \tau = X,\\ T = (vec, X, O) &\Rightarrow \tau = X,\\ T = (mat, C, R) &\Rightarrow \tau = C,\\ T = (mat, C, O_C, R, O_R) &\Rightarrow \tau = C,\\ T = f &\Rightarrow \tau = image(f),\\ T = (list, T') &\Rightarrow \tau = T'.\end{aligned}$$

5.2.3. *Context Type List Sub-List Selection.*

$$(281)\qquad (m, A, B)\triangle(list, \tau) \rightsquigarrow (m, A, B)\triangle sub((list, \tau), p_s).$$

5.2.4. *Context Type List Element Selection.*

$$(282)\qquad (m, A, B)\triangle(list, \tau) \rightsquigarrow select((list, \tau), p_s).$$

5.2.5. *Automated Exist Context Construction.*

$$(283)\qquad \exists X \in \lceil ObjectClass\rfloor \rightsquigarrow \exists_{Sys}(X).$$

5.2.6. *Vector Construction.*

$$\begin{aligned}(284)\quad (m, A, B)\triangle AppliedValue &\rightsquigarrow (build_{vector}, A, B)\triangle AppliedVector^\star,\\ \exists_{Sys}(B)\triangle ObjectList &\wedge AppliedValue = (val, C) \Rightarrow\\ dim(AppliedVector^\star) &= length(ObjectList) \wedge\\ AppliedVector^\star &= (vec^\star, C).\end{aligned}$$

5.2.7. *Matrix Construction.*

$$\begin{aligned}(285)\ (m, A, B)\triangle AppliedVector &\rightsquigarrow (build_{matrix}, A, B)\triangle AppliedMatrix^\star,\\ \exists_{Sys}(B)\triangle ObjectList &\wedge AppliedVector = (vec, C) \Rightarrow\\ AppliedMatrix^\star &= (mat^\star, B, C).\end{aligned}$$

5.2.8. *Vector Extraction.*

$$\begin{aligned}(286)\qquad (m, A, B)\triangle(mat, C, R) &\rightsquigarrow (select_{col}, A, B)\triangle(vec, R)\\ (vec, R) &= select_{col}(mat, C, R),\\ (287)\qquad (m, A, B)\triangle(mat, C, R) &\rightsquigarrow (select_{row}, A, R)\triangle(vec, C)\\ (vec, C) &= select_{row}(mat, C, R).\end{aligned}$$

5.2.9. *Value Extraction.*

$$\begin{aligned}(288)\qquad (m, A, B)\triangle(vec, S) &\rightsquigarrow (select_{comp}, A, S)\triangle(val, S),\\ (val, S) &= select_{comp}(vec, S).\end{aligned}$$

5.2.10. *Context Type List Construction.*

$$(289)\qquad \begin{array}{rcl}(m,A,B)\triangle(vec,S) & \rightsquigarrow & (build_{list},A,B)\triangle\big((m,A,S)\triangle(val,S)\big)\\ (val,S) & = & select_{comp}((vec,S),p_s).\end{array}$$

5.3. Example Selection Predicates. We will now briefly discuss the issue of selection predicates, as these can span a broad number of different applications. Needless to say, a selection predicate operates on a list, and can thus make statements about the items of a list.

As our indexed set lists can be mapped to n-tuples (using the $\iota\tau$ function in Section 2.3, Chapter 1), we will use a sub-list relation $\sqsubseteq$:

$$(290)\qquad \begin{array}{c}(a_1,...,a_n)\sqsubseteq(b_1,...,b_m)\qquad\qquad\Leftrightarrow\\ \Leftrightarrow\quad \forall i,1\leq i\leq n\,\exists j,1\leq j\leq m:a_i=b_j\wedge n\leq m.\end{array}$$

Formulated as a unary list relation, we could define the following selection predicate:

$$(291)\qquad \begin{array}{rcl}p_s & \sqsubseteq & (v_1,...,v_n)\\ p_s(v_i) & \textit{holds, iff} & v_i\neq 0,\,\forall i,1\leq i\leq n,\end{array}$$

which returns all non-zero elements of a list.

The open nature of a selection *predicate* also allows considerations about the entire list to be used in the decision on an individual item.

As an example, we can define a selection predicate for selecting the ten highest numbers from a set with the following predicate as an unary list relation.

$$(292)\qquad \begin{array}{rcl}p_s & \sqsubseteq & (v_1,...,v_n)\\ p_s(v_i) & \textit{holds, iff} & v_i\geq r,\,\forall i,1\leq i\leq n,\end{array}$$

where we choose r, such that the following holds:

$$(293)\qquad \|\{v'|v'\in\{v_1,...v_n\}\wedge v'\geq r\}\| \;=\; \begin{cases}10 & \textit{if } n\geq 10.\\ n & \textit{otherwise.}\end{cases}$$

Predicates of this manner can be used to turn context type lists into top-n recommendations.

CHAPTER 5

Causality, Rules, Similarity and Traversal

1. A Type Causality Category

As described in Chapter 4, we can now infer new typed context types, but as we will need several expanding steps to obtain useful results, we also need some means of describing the causality - the reason why we can deduce any given context type.

To describe this, we will model the "history" of all context types and their correspoding type concepts in a category. We will use a new type of categorical objects, namely a composite of a context type and a type concept. As we want to retain the capability to underspecify, but require the type concept for deduction, we will now define an *empty* type concept, which can be used if no type concept has been specified or deduced.

Definition 1.1 (Empty Type Concept). An empty type concept, denoted by $\emptyset_{TC}$, is defined as follows:

$$\emptyset_{TC} = \emptyset. \tag{294}$$

The empty set satisfies all requirements to be a type concept, but has no attributes or properties.

We can now define our notion of a typed context type.

Definition 1.2 (Typed Context Type). A typed context type is a combination of a context type and the assigned type concept.

$$\lceil TypedContextType \rfloor = \lceil ContextType \rfloor \times \lceil TypeConcept \rfloor. \tag{295}$$

The following rules hold for a well formed typed context type:

$$\begin{aligned} TypedContextType &= (C, T) \\ C \nabla \mu &\Rightarrow T = \emptyset_{TC}, \\ \neg(C \nabla \mu) &\Rightarrow T = \mu(C), \\ C &\triangle T, \end{aligned} \tag{296}$$

for any typing function μ.

With these definitions, we can define our new category of typed context type causality $\mathcal{TCT}_?$.

Definition 1.3 (Typed Context Type Causality Category). We define a category, which hosts typed context types as objects, and has an arrow $A \to B$ if A part of the premises, which have led to the deduction of B, meaning that A *and*

any other combination of statements X have led to B.

$$
\begin{array}{lrcl}
(297) & obj(\mathcal{TCT}_?) & \subseteq & \lceil TypedContextType \rfloor \\
(298) & mor(\mathcal{TCT}_?) & \subseteq & \lceil TypedContextType \rfloor^2 \\
(299) & A \to B & \Leftrightarrow & (A,B) \in mor(\mathcal{TCT}_?) \\
(300) & A \to B & \Leftrightarrow & A \wedge X \rightsquigarrow B \\
(301) & \forall A \in obj(\mathcal{TCT}_?) : id_A = (A,A) & \in & mor(\mathcal{TCT}_?) \\
(302) & A \to B \circ C \to D & \Rightarrow & A \to D \textit{ iff } B = C \\
(303) & & &
\end{array}
$$

2. A Rule Category

We will now model a category, which collects our rules $\circ$, $\bowtie$, $\Diamond$, $\triangle$, ∇, $\triangleright$ and $\triangleleft$, and the corresponding rules, which hold for the objects of type $\lceil TypedContextType \rfloor$. We want this category to be an extension of the $\mathcal{TCT}_?$ category, so we will first model the typing rules $\triangle$ and ∇ for typed context types, denoted by $\triangle_{TCT}$ and ∇_{TCT}.

Definition 2.1 (Typed Context Type Typing Rules). We define the rules $\triangle_{TCT}$ and ∇_{TCT} as unary relations and variants of the original typing rules $\triangle$ and ∇.

$$
\begin{array}{lrcl}
(304) & \triangle_{TCT} & \subseteq & \lceil TypedContextType \rfloor \\
(305) & (T,TC)\triangle_{TCT} & \textit{holds, iff} & T \neq \emptyset_{TC}, \\
(306) & \nabla_{TCT} & \subseteq & \lceil TypedContextType \rfloor \\
(307) & (T,TC)\nabla_{TCT} & \textit{holds, iff} & T = \emptyset_{TC}.
\end{array}
$$

We can now present our categorical modeling.

DEFINITION 2.2 (Typed Context Type Rule Category). The category of rules for typed context types $\mathcal{TCT}_!$ is defined as follows.

$$obj(\mathcal{TCT}_!) \subseteq \lceil TypedContextType \rfloor \tag{308}$$

$$\lceil Rules \rfloor = \{\circ, \bowtie, \Diamond, \triangle_{TCT}, \nabla_{TCT}, \triangleright, \triangleleft\} \tag{309}$$

$$mor(\mathcal{TCT}_!) \subseteq \lceil TypedContextType \rfloor^2 \times \lceil Rules \rfloor \times \tag{310}$$

$$\times \{true, false\} \tag{311}$$

$$A \xrightarrow[b]{Rule} B \Leftrightarrow (A, B, Rule, b) \tag{312}$$

$$\forall A \in obj(\mathcal{TCT}_!) : \begin{cases} A \xrightarrow[true]{\triangle} A & iff\ A \triangle_{TCT} \\ A \xrightarrow[true]{\nabla} A & iff\ A \nabla_{TCT} \end{cases} \tag{313}$$

$$\forall A, B \in obj(\mathcal{TCT}_!) : A \bowtie B \Rightarrow A \xrightarrow[true]{\bowtie} B \tag{314}$$

$$\forall A, B \in obj(\mathcal{TCT}_!) : A \circ B \Rightarrow A \xrightarrow[true]{\circ} B \tag{315}$$

$$\forall A, B \in obj(\mathcal{TCT}_!) : A \Diamond B \Rightarrow A \xrightarrow[true]{\Diamond} B \tag{316}$$

$$\forall A, B \in obj(\mathcal{TCT}_!) : A \triangleright B \Rightarrow A \xrightarrow[true]{\triangleright} B \tag{317}$$

$$\forall A, B \in obj(\mathcal{TCT}_!) : A \triangleleft B \Rightarrow A \xrightarrow[true]{\triangleleft} B\ . \tag{318}$$

We have the following translations between morphisms:

$$A \xrightarrow[true]{\triangle} A \Leftrightarrow A \xrightarrow[false]{\nabla} A \tag{319}$$

$$A \xrightarrow[true]{\nabla} A \Leftrightarrow A \xrightarrow[false]{\triangle} A \tag{320}$$

$$A \xrightarrow[true]{\triangleright} B \Rightarrow A \xrightarrow[false]{\triangleleft} B \tag{321}$$

$$A \xrightarrow[true]{\triangleleft} B \Rightarrow A \xrightarrow[false]{\triangleright} B \tag{322}$$

$$A \xrightarrow[true]{\bowtie} B \Leftrightarrow A \xrightarrow[true]{\triangle} A \wedge B \xrightarrow[true]{\triangle} B \tag{323}$$

$$A \xrightarrow[true]{\Diamond} B \Leftrightarrow A \xrightarrow[true]{\circ} B \wedge A \xrightarrow[true]{\bowtie} B \tag{324}$$

$$A \xrightarrow[false]{\Diamond} B \Leftrightarrow A \xrightarrow[false]{\circ} B \vee A \xrightarrow[false]{\bowtie} B \tag{325}$$

$$A \xrightarrow[true]{\triangleright} B \Leftrightarrow A \xrightarrow[true]{\circ} B \wedge A \xrightarrow[false]{\bowtie} B \tag{326}$$

$$A \xrightarrow[false]{\triangleright} B \Leftrightarrow A \xrightarrow[false]{\circ} B \vee A \xrightarrow[true]{\bowtie} B \tag{327}$$

$$A \xrightarrow[true]{\lhd} B \quad \Leftrightarrow \quad A \xrightarrow[false]{\circ} B \ \wedge \ A \xrightarrow[false]{\bowtie} B \tag{328}$$

$$A \xrightarrow[false]{\lhd} B \quad \Leftrightarrow \quad A \xrightarrow[true]{\circ} B \ \vee \ A \xrightarrow[true]{\bowtie} B \ , \tag{329}$$

and define the composition as follows:

$$comp : mor(\mathcal{TCT}_!) \times mor(\mathcal{TCT}_!) \ \rightarrow \ mor(\mathcal{TCT}_!) \tag{330}$$

$$(\ A \xrightarrow[b]{Rule} A \ , \ A \xrightarrow[b]{Rule} A \ ) \ \mapsto \ A \xrightarrow[b]{Rule} A \tag{331}$$

$$(\ A \xrightarrow[b]{\triangle} A \ , \ A \xrightarrow[true]{\bowtie} B \ ) \ \mapsto \ A \xrightarrow[b]{\bowtie} B \tag{332}$$

$$(\ A \xrightarrow[true]{\bowtie} B \ , \ B \xrightarrow[b]{\triangle} B \ ) \ \mapsto \ A \xrightarrow[b]{\bowtie} B \tag{333}$$

$$(\ A \xrightarrow[false]{Rule} A \ , \ A \xrightarrow[false]{\bowtie} B \ ) \ \mapsto \ A \xrightarrow[A \circ B]{\rhd} B \tag{334}$$

$$(\ A \xrightarrow[false]{\bowtie} B \ , \ B \xrightarrow[false]{Rule} B \ ) \ \mapsto \ A \xrightarrow[A \circ B]{\rhd} B \tag{335}$$

$$(\ A \xrightarrow[true]{\triangle} A \ , \ A \xrightarrow[b]{\circ} B \ ) \ \mapsto \ A \xrightarrow[b \wedge A \bowtie B]{\Diamond} B \tag{336}$$

$$(\ A \xrightarrow[b]{\circ} B \ , \ B \xrightarrow[true]{\triangle} B \ ) \ \mapsto \ A \xrightarrow[b \wedge A \bowtie B]{\Diamond} B \tag{337}$$

$$(\ A \xrightarrow[true]{\nabla} A \ , \ A \xrightarrow[b]{\circ} B \ ) \ \mapsto \ A \xrightarrow[b]{\rhd} B \tag{338}$$

$$(\ A \xrightarrow[b]{\circ} B \ , \ B \xrightarrow[true]{\nabla} B \ ) \ \mapsto \ A \xrightarrow[b]{\rhd} B \tag{339}$$

$$(\ A \xrightarrow[b]{Rule} A \ , \ A \xrightarrow[b']{\rhd} B \ ) \ \mapsto \ A \xrightarrow[b']{\rhd} B \tag{340}$$

$$(\ A \xrightarrow[b']{\rhd} B \ , \ B \xrightarrow[b]{Rule} B \ ) \ \mapsto \ A \xrightarrow[b']{\rhd} B \tag{341}$$

$$(\ A \xrightarrow[b]{Rule} A \ , \ A \xrightarrow[b']{\lhd} B \ ) \ \mapsto \ A \xrightarrow[b']{\lhd} B \tag{342}$$

$$(\ A \xrightarrow[b']{\lhd} B \ , \ B \xrightarrow[b]{Rule} B \ ) \ \mapsto \ A \xrightarrow[b']{\lhd} B \tag{343}$$

$$(\ A \xrightarrow[b]{R} B \ , \ B \xrightarrow[b]{R'} C \ ) \ \mapsto \ \begin{cases} A \xrightarrow[ARC]{R} C & ... \quad R = R' \\ A \xrightarrow[A \Diamond C]{\Diamond} C & ... \quad R \neq R', \end{cases} \tag{344}$$

for any $A, B, C \in obj(\mathcal{TCT}_!)$, $A \neq B \neq C$.

The composition can be computed for any combination of arrows, such that the following holds:

$$A \xrightarrow[b]{r} B \ \circ' \ C \xrightarrow[b']{r'} D \qquad \mapsto \tag{345}$$

$$\mapsto \ \begin{cases} comp(\ A \xrightarrow[b]{r} B \ , \ C \xrightarrow[b']{r'} D \ ) & ... \ B = C \\ \emptyset_{mor(\mathcal{TCT}_!)} & otherwise, \end{cases}$$

where $\emptyset_{mor(\mathcal{TCT}_!)}$ is a designated element not used for any other purpose than denoting an invalid composition.

This categorical definition uses a rather unusual composition function. The basic design of this function becomes evident if we study it in three parts. Statement 331 handles all compositions of the form $id_A \circ' id_A$.

From Statement 332 to Statement 343 we have the composition rules for any composition of an arrow with any identity arrow, such that the categorical composition requirement holds:

$$id_A \circ' A \to B = A \to B \circ' id_B. \tag{346}$$

The final rule, as in Statement 344, expresses compositions of arrows without identity arrows. If we are compositing two arrows with identical rules, we use this rule in the composition. If we want to composit two arrows with different rules, we simply use the $\Diamond$ rule, as it collects both context type and type concept considerations.

3. Context Similarity

We will now craft another category. The category $\mathcal{ACS}$ of abstract context entity similarity can be used to describe and partially compute similarity ratings between abstract context entities from different typed context types. As similarity always evolves around a mathematical domain, we will craft $\mathcal{ACS}$ categories for a given monoid, a semigroup with a zero and one element, denoted by K.

We have the following requirements regarding K:

$$K = (K, \times_k), \tag{347}$$

$$\times_k : K \times K \to K, \tag{348}$$

$$\forall a, b, c \in K : a \times_k (b \times_k c) = (a \times_k b) \times_k c, \tag{349}$$

$$\exists 1_k \in K : k \times_k 1_k = 1_k \times_k k = k, \tag{350}$$

$$\exists \emptyset_k \in K : k \times_k \emptyset_k = \emptyset_k \times_k k = \emptyset_k. \tag{351}$$

Furthermore, we will define a function template for context entity similarity , denoted by ρ_k.

Definition 3.1 (An Abstract Context Entity Similarity Function Template). For context pairs, we have a function ρ_k for affinity or distance measurement.

$$\rho_k : \lceil Context \rfloor \times \lceil Context \rfloor \to k. \tag{352}$$

In most systems, it will not be possible to compute the similarity between all context types using a single function. In this case, the ρ function will serve as a wrapper for the individual similarity functions. But as we also have the requirement of a single K monoid across one category, an $\mathcal{ACS}$ category must be based on an initial selection of which context types will be considered for the similarity category on instance level.

Definition 3.2 (The Category of Abstract Context Entities with Similarity). An $\mathcal{ACS}$ context similarity category collects context entities, which are instances of context types, for which we can compute some means of similarity, and which

return a value within K. Mathematically expressed, we can construct a context type selection as follows:

$$\forall A, B \in \lceil Context \rfloor \quad , \quad T(A) = \tau_1, T(B) = \tau_2 : \tag{353}$$

$$\tau_1 \triangle \mu(\tau_1) \wedge \tau_2 \triangle \mu(\tau_2) \rightsquigarrow C \triangle T \wedge T \in K \quad \Leftrightarrow \quad \tau_1, \tau_2 \in TCTSelection \wedge \tag{354}$$

$$\wedge \quad C \ni TCTSelection, \tag{355}$$

$$TCTSelection \quad \subseteq \quad \lceil ContextType \rfloor, \tag{356}$$

for any typing function μ.

Our objects for a $\mathcal{ACS}$ category will be abstract context entities, with a context type from a suitable selection.

$$obj(\mathcal{ACS}) \quad = \quad \{C | T(C) \in TCTSelection\}. \tag{357}$$

The category $\mathcal{ACS}$ of abstract context entities with similarity is defined as follows:

$$obj(\mathcal{ACS}) \quad = \quad \{C | T(C) \in TCTSelection\}, \tag{358}$$

$$mor(\mathcal{ACS}) \quad \subseteq \quad \lceil Context \rfloor \times \lceil Context \rfloor \times K, \tag{359}$$

$$\forall c_1, c_2 \in obj(\mathcal{ACS}) \quad : \quad (c_1, c_2, \rho_k(c_1, c_2)) \in mor(\mathcal{ACS}) \tag{360}$$

$$\exists (A, B, \omega) \in mor(\mathcal{ACS}) \quad \Leftrightarrow \quad A \xrightarrow[\rho_k(A,B)]{} B, \tag{361}$$

$$\forall A, B, C \in obj(\mathcal{ACS}) \quad : \quad A \xrightarrow[\times_k(\omega_1, \omega_2)]{} C \textit{ iff } A \xrightarrow[\omega_1]{} B \wedge \tag{362}$$

$$\wedge \ B \xrightarrow[\omega_2]{} C$$

$$\forall A \in obj(\mathcal{ACS}) \quad : \quad A \xrightarrow[1_k]{id_A} A. \tag{363}$$

(364)

This allows multiple arrows between contexts, each representing a different approach. This is a general definition which leaves the nature of the rating k and the connecting operation $\times_k$ open, to allow different ρ functions.

3.1. Possible Application of the $\mathcal{ACS}$ Category. Possible appliations for the $\mathcal{ACS}$ category could be in a translation role between fuzzy context models and predicate logic based inferencing systems based on formalized context [**3**].

If we have an assignment from the contexts of a category $\mathcal{AC}$ to general, abstract context entities, we can use this as a bridge to other systems using this notion.

A function template could be defined as

$$ext : obj(\mathcal{AC}) \rightarrow \lceil Context \rfloor. \tag{365}$$

We could now assign several contexts within $\mathcal{AC}$ to identical external context entities. This could be a decision based on their affinity in a category $\mathcal{ACS}$.

After mapping to external context, which is part of an inferencing system, we can draw conclusions in this inferencing system, map back to $\mathcal{AC}$ and use the similarities in the $\mathcal{ACS}$ category as a measurement for the reliability of the conclusion.

4. Context Traversal

We now define a category for specifying when a traversal along a possible composition may, or may not be done. This means that we will define a category, which is very similar to the $\mathcal{AC}$, $\mathcal{CT}$ or $\mathcal{CG}$ categories, but has a boolean weight on

every arrow. This boolean weight expresses wheather or not it is allowed to "use or traverse" this arrow.

This can be used to express various things:

- Can the context composition can be computed?
- Can a context similarity be computed?
- Is this combination realized in a system?
- Any other "yes or no" question about an arrow in $\mathcal{AC}$.

This means that such a category can also contain the truth value for any specific query to a $\mathcal{TCT}_!$ rule category. To be used for this purpose, we will define the categorical arrow composition as an $\wedge$ operator for the boolean weight of the arrows. This loosens the constraints of the original $\mathcal{TCT}_!$ rule category, but composition can be used additionally to a full connectivity derived from $\mathcal{TCT}_!$.

Definition 4.1 (The Category of Abstract Context Entity Traversal). The category $\mathcal{ACT}$ of abstract context entity traversal is defined as follows:

$$obj(\mathcal{ACT}) = \lceil Context \rfloor, \tag{366}$$

$$mor(\mathcal{ACT}) \subseteq \lceil Context \rfloor \times \lceil Context \rfloor \times \{true, false\}, \tag{367}$$

$$\forall c_1, c_2 \in obj(\mathcal{ACT}) : c_2 \in \circ_{AC}(c_1) \Leftrightarrow (c_1, c_2, b) \in mor(\mathcal{ACT}) \tag{368}$$

$$\exists (A, B, b) \in mor(\mathcal{ACT}) \Leftrightarrow A \xrightarrow[b]{} B, \tag{369}$$

$$\forall A, B, C \in obj(\mathcal{ACT}) : A \xrightarrow[b_1 \wedge b_2]{} C \textit{ iff } A \xrightarrow[b_1]{} B \wedge \wedge\ B \xrightarrow[b_2]{} C \tag{370}$$

$$\forall A \in obj(\mathcal{ACT}) : A \xrightarrow[b]{id_A} A. \tag{371}$$

Note that the weight of an arrow is explicitly specified according to the desired usage of the category.

We can define analogous categories for context type traversal, denoted by $\mathcal{CTT}$, and context group traversal, denoted by $\mathcal{CGT}$ by using context types or context groups and their corresponding composition function, but an otherwise identical category definition as in Definition 4.1.

CHAPTER 6

Typed Objects

1. What Are Object Types?

Up to now, we have concentrated of the underlying type concept of a context type, leading to typed context types. But at some point, it becomes necessary to specify more than just the context type's type concept, leading to the new notion of typed objects and typed object classes.

If we want to specify the type of an object class, our initial type concepts will not suffice to describe a real system. We need to expand our type notions to include various forms of content, or more generally, digital media documents, entities from knowledge representations and other forms of common system data or information. This means that we must now define a more general understanding of types.

1.1. Extended Primitives and Type Concepts. To describe objects, we need an initial set of basic types. As before with type concepts (see Chapter 3), we begin by defining a few primitives. In order to describe rich system environments, we will use a very extensive definition of primitive types.

Definition 1.1 (Extended Primitive Types). We now define the set of all extended primitive types,

$$\lceil ExtPrimitive \rfloor = \{Boolean, Integer, Real, String, URI\}. \tag{372}$$

We also need the notion of values and object lists of extended primitives.

Definition 1.2 (Extended Values). If we assume that $i :: T$ denotes that an instance i has type T, we can define an extended value as follows.

$$\lceil XValue \rfloor_T = \{i | i :: T\}, \tag{373}$$

where T is any extended primitive type,

$$T \in \lceil ExtPrimitive \rfloor. \tag{374}$$

Definition 1.3 (Extended Object Lists). An extended object list is an object list, which is based on extended primitives. If we again assume that $i :: T$ denotes that an instance i has type T, it is defined as follows.

$$\lceil XList \rfloor_T = \left\lceil \langle^n \lceil XValue \rfloor^n \rangle \right\rfloor, \tag{375}$$

where T is any extended primitive type,

$$T \in \lceil ExtPrimitive \rfloor, \tag{376}$$

and the following holds for $\lceil XList \rfloor_T$:

$$\lceil XList \rfloor_T = \langle L \rangle, \tag{377}$$

$$\forall i, 1 \leq i \leq n \quad : \quad \langle L \rangle.i :: T. \tag{378}$$

We will now subsume our definitions on extended primitives by presenting an extended definition of the type concepts, by replacing our initial notion of values and lists by the extended variants.

Definition 1.4 (Extended Type Concepts). The set of extended type concept specifications $\lceil XTypeConcept \rfloor$ is a variant of the set $\lceil XTypeConcept \rfloor$, but introduces values and lists based on extended primitives.

$$\begin{aligned}(379)\quad \lceil XTypeConcept \rfloor \;=\; \big\{ & T \,|\, T \in \lceil XValue \rfloor \vee T \in \lceil XList \rfloor \\ \vee\; & T \in \lceil Vector \rfloor \vee T \in \lceil Matrix \rfloor \\ \vee\; & T \in \lceil Assignment \rfloor \\ \vee\; & T \in \lceil AppliedValue \rfloor \vee T \in \lceil BoundValue \rfloor \\ \vee\; & T \in \lceil AppliedVector \rfloor \vee T \in \lceil BoundVector \rfloor \\ \vee\; & T \in \lceil AppliedMatrix \rfloor \vee T \in \lceil BoundMatrix \rfloor \\ \vee\; & T \in \lceil ObjectList \rfloor \vee T \in \lceil CTList \rfloor \big\}.\end{aligned}$$

Now that we have an extended understanding of type concepts, let us turn to higher level types.

1.2. Media Types. On object class level, we must consider different forms of media. Even though there are standards on metadata, such as elaborated by the Dublin Core Metadata Initiative[1] (see [**1**]), we will take a much more general approach by using a very crude taxonomy of media documents.

We will use an initial specification of different types of media:

- Texts
- Images
- Audio
- Video

But we will further refine these by assigning a few properties:

- Embedding: Linked (HTML) / Single (RTF)
- Content Editing: Editable (PDF) / Protected (PS)
- Content Behaviour: Interactive (VRML) / Static (BMP)
- Author: Authenticated / Stated / Unknown
- Source: Citation Details / URI / Description / Unknown

We can now craft a few sets, which serve as a basis for our mathematical definitions of media types.

Definition 1.5 (Media Types). To define our type notion for media documents, we initially define six sets of properties.

$$\begin{aligned}(380)\quad Content &= \{Text, Image, Audio, Video\},\\ (381)\quad Embedding &= \{Linked, Single\},\\ (382)\quad Editing &= \{Editable, Protected\},\\ (383)\quad Behaviour &= \{Interactive, Static\},\\ (384)\quad Author &= \{Authenticated, Stated, Unknown, Variable\},\\ (385)\quad Source &= \{CitationDetails, URI, Description, Unknown\}.\end{aligned}$$

[1] See `http://dublincore.org`

We now define the set $\lceil MediaType \rfloor$ of media types, which represent tokens for media document sets, based on our property sets.

$$\lceil MediaType \rfloor = Content \times Embedding \times Editing \times \times Behaviour \times Author \times Source. \tag{386}$$

We can now assume that many HTML documents belong to the following media type:

$$HTML :: (Text, Linked, Protected, Interactive, Variable, URI). \tag{387}$$

Let us consider the individual properties of this example:

- Text: Textual Media Form
- Linked: uses or allows links.
- Protected: without annotations, we can consider web content to be protected.
- Interactive: as we are concerned with describing systems, we do not consider links to be interactive, but the ability to fill in forms, using text fields, radial buttons and other GUI elements are considered to be interactive.
- Author Variable: There are standards for the author of an HTML page, but this field is not mandatory.
- Source URI: every HTML document in the WWW has a URI.

1.3. Knowledge Descriptions. In the days of the semantic web[2], it is impossible to disregard different knowledge structures. But the progress in the field of semantic web application development and exploitation is accompanied by many different standards on how to *describe* knowledge representations.

As any attempt to formulate this in type definitions would result in very specialized definitions unsuitable for a general model, any types necessary for objects, which are related to ontologies, should be crafted system dependent using our notion of *compound types*, which we will describe below. Being similar to frames, these will suffice to *"tie in"* to different forms of knowledge representation.

2. Compound Types

As a step towards representing frames, we will introduce the notion of compound types. Compound types allow us to *build* type definitions, which are based on *multiple, labeled extended type concepts or media types.*

Initially, we define the notion of a compound element, which serves as an atomic element of the compound types.

Definition 2.1 (Compound Elements). A compound element is a labeled extended type concept or media type.

$$\lceil CompoundElement \rfloor = \lceil Identifier \rfloor \times \times (\lceil XTypeConcept \rfloor \cup \lceil MediaType \rfloor). \tag{388}$$

We can now define our compound types, which are composited from compound elements.

[2]See `http://www.w3c.org/2001/sw/`

Definition 2.2 (Compound Types). A compound type is either one compound element, or a collection of compound elements with an additional main label. Mathematically, this is defined over a set of tuples,

$$\begin{aligned} (389) \quad \lceil CompoundType \rfloor &= \lceil CompoundElement \rfloor \\ &\cup \left(\lceil Identifier \rfloor \times \lceil CompoundElement \rfloor^2 \right) \\ &\cup \quad ... \\ &\cup \left(\lceil Identifier \rfloor \times \lceil CompoundElement \rfloor^n \right), \end{aligned}$$

where n is any positive non-infinite integer value,

$$(390) \qquad n \in N \wedge n > 2 \wedge \exists m \in N : n < m.$$

We can now turn to the rules and statements on compound types.

2.1. Compound Type Rules. Initially, we need the ability to type objects. We will now present a definition for specifying that an *object* is an *instance* of a given compount type.

Definition 2.3 (Typing Objects). If we have an object o, and a compound type T, we will use $o :: T$ to denote the following:

$$\begin{aligned} (391)\ o :: T \quad \textit{iff} \quad & T = (Label, XTypeConcept) \wedge o :: XTypeConcept \quad \vee \\ & \vee \quad T = \Big(Label, \big((l_1, T_1), ..., (l_n, T_n) \big) \Big) \wedge o = (o_1, ..., o_n) \\ & \wedge \quad \forall i \in N, 2 \leq i \leq n : o_i :: T_i \quad \vee \\ (392) \qquad & \vee \quad T = (Label, MediaType) \wedge o :: MediaType \quad \vee \\ & \vee \quad MT = \Big(Label, \big((l_1, MT_1), ..., (l_n, MT_n) \big) \Big) \wedge o = (o_1, ..., o_n) \\ & \wedge \quad \forall i \in N, 2 \leq i \leq n : o_i :: MT_i, \end{aligned}$$

for any extended type concept T_i or media type MT_i.

When dealing with compound types, we may want to know which compound types *break down* to extended type concepts. This is of particular interest, as we may then be able to apply our deduction rules from Chapter 4 on compound types, and any object, which is type in this manner.

Definition 2.4 (Nested Single Type Concepts). We define a rule , denoted by $\sqsubseteq$, which states that a compound type consists of *only one specific labeled type concept*, over a binary relation.

$$\begin{aligned} (393) \qquad \sqsubseteq \quad & \subseteq \quad \lceil XTypeConcept \rfloor \times \lceil CompoundType \rfloor \\ T \sqsubseteq C \quad & \textit{holds, iff} \quad C = (Label, T') \wedge T = T'. \end{aligned}$$

Analogously, we define a rule $\sqsubset$, which states that a compound type consists of *any one labeled type concept*, over an unary relation.

$$\begin{aligned} (394) \qquad \sqsubset \quad & \subseteq \quad \lceil CompoundType \rfloor \\ C \sqsubset \quad & \textit{holds, iff} \quad C = (Label, T) \wedge T \in \lceil TypeConcept \rfloor. \end{aligned}$$

We can now apply our notion of compound types to our context model.

3. Typed Object Classes

Initially, our notion of object classes was independent from classical modeling principles, which are usually based on data types or classes in an object-oriented sense. Instead, we have defined them as mere collections of objects, independent from the nature of the classification, which has led to the partition of objects necessary for a clean conception with a one-to-one dependency between types and instances.

We now introduce the notion of typed object classes, which leads to us back to the usual approach of using data-types or frames as descriptive templates for all objects, which belong to this class. Conversely, we can see a compound type as the basis for the decision, to which object class an object must be assigned, or to which object classes it might belong in more complex systems.

Definition 3.1 (Typed Object Classes). A typed object class is an object class and a compound type, which specifies the type of all objects, which belong to the object class. This is defined as,

$$\lceil TypedObjectClass \rfloor = \lceil ObjectClass \rfloor \times \lceil CompoundType \rfloor, \tag{395}$$

such that the following constraint is met:

$$\forall (C, T) \in \lceil TypedObjectClass \rfloor : o \in C \Leftrightarrow o :: T. \tag{396}$$

Now that we have an understanding of typed object classes, we can draft another typing function template μ_{OC} for object classes.

Definition 3.2 (Object Class Typing Function Template). We now present a function template for a typing function on object classes. The underlying type representation is that of a compound type.

$$\mu_{OC} : \lceil ObjectClass \rfloor \quad \rightarrow \quad \lceil CompoundType \rfloor. \tag{397}$$

With these two definitions, we can conclude our initial model for typing objects and typed object classes, and can now turn to describing the workflow, information flow and processing in a system.

CHAPTER 7

Workflow and Processing

1. Context Types with Object Class View

Up to now, our categories have been based on context entities, context types, context groups and typed context types. We will now craft categories based on object classes.

An alternate way of viewing our basic context type category $\mathcal{CT}$ is to draw an arrow between those object classes, which are connected by a method, forming a context type. Furthermore, we will label this arrow with the method of the respective context type.

Definition 1.1 (Object Class Context Type View). We define a category $\mathcal{CT}_T^{OC}$, which represents a set of context types T using object classes as main objects.

$$obj(\mathcal{CT}_T^{OC}) \subseteq \lceil ObjectClass \rfloor \tag{398}$$

$$mor(\mathcal{CT}_T^{OC}) \subseteq \lceil ObjectClass \rfloor \times \lceil ObjectClass \rfloor \times \times \lceil Method \rfloor \tag{399}$$

$$\exists(A,B,m) \in mor(\mathcal{CT}_T^{OC}) \Leftrightarrow A \xrightarrow{m} B \tag{400}$$

$$\exists(m,A,B) \in T \Rightarrow A \in obj(\mathcal{CT}_T^{OC}) \wedge B \in obj(\mathcal{CT}_T^{OC}) \wedge$$

$$\wedge \quad A \xrightarrow{m} B \in mor(\mathcal{CT}_T^{OC})$$

$$\forall A,B,C \in obj(\mathcal{CT}_T^{OC}) : A \xrightarrow{m_\circ} C \text{ iff } \tag{401}$$

$$A \xrightarrow{m} B \wedge B \xrightarrow{m'} C .$$

where the method $m_\circ$ denotes a possible composition.

The object class view on context types can give a summary of the dependencies amongst the object classes, and is a good categorical basis for the visualization of models (see Figure 1).

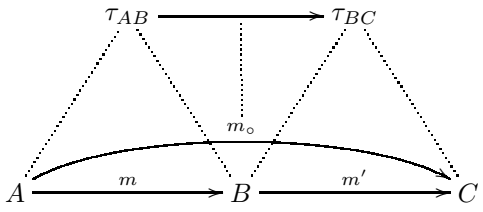

Figure 1. Object Class View on Context Types

1.1. Typed Object Classes. We can craft a very similar category, based on typed object classes. If we briefly recapture our definition of a typed object class,

$$\lceil TypedObjectClass \rfloor = \lceil ObjectClass \rfloor \times \lceil CompoundType \rfloor, \tag{402}$$

we can now present our categorical model.

DEFINITION 1.2 (Typed Object Class View on Context Types). For a given set of typed context types T and an object class typing function according to the function template μ_{OC}, we define the category $\mathcal{CT}_T^{TOC}$ of the typed object class view as follows.

$$obj(\mathcal{CT}_T^{TOC}) \subseteq \lceil TypedObjectClass \rfloor \tag{403}$$

$$\begin{aligned} mor(\mathcal{CT}_T^{TOC}) &\subseteq \lceil TypedObjectClass \rfloor \times \\ &\times \lceil TypedObjectClass \rfloor \times \\ &\times \lceil Method \rfloor \times \lceil TypeConcept \rfloor \end{aligned} \tag{404}$$

$$\exists(A, B, m, t) \in mor(\mathcal{CT}_T^{TOC}) \Leftrightarrow A \xrightarrow[t]{m} B \tag{405}$$

$$\begin{aligned} \exists(m, A, B)\triangle t \in T \Rightarrow\ & a = (A, \mu_{OC}(A)) \in obj(\mathcal{CT}_T^{TOC}) \\ \wedge\ & b = (B, \mu_{OC}(B)) \in obj(\mathcal{CT}_T^{TOC}) \wedge \\ \wedge\ & a \xrightarrow[t]{m} b \in mor(\mathcal{CT}_T^{TOC}) \end{aligned} \tag{406}$$

$$\begin{aligned} \forall A, B, C \in obj(\mathcal{CT}_T^{TOC}) :\ & A \xrightarrow[t_\circ]{m_\circ} C \ \textit{iff} \\ & A \xrightarrow[t]{m} B \ \wedge\ B \xrightarrow[t']{m'} C\ , \end{aligned} \tag{407}$$

where the following holds for $m_\circ$ and $t_\circ$:

$$(m, A, B)\triangle t \wedge (m', B, C)\triangle t' \rightsquigarrow (m_\circ, A, C)\triangle t_\circ \quad \vee \tag{408}$$

$$\vee \qquad m_\circ = NullMethod \wedge t_\circ = \emptyset_{TC}, \tag{409}$$

and the method $NullMethod$ denotes that no composition is possible.

1.2. The Graph Interpretation. Mathematically, our object class view on context types can be seen as a directed multi-graph in function representation. This interpretation allows us to compute the transitive hull or other graph properties, which we can exploit for model analysis.

Our considerations on graph representations are very useful on our $\mathcal{CT}_T^{TOC}$ category, but can be applied to our other context categories.

1.2.1. *Transitive Hull.* The transitive hull allows us to check a model for basic composability across the entire model. This can be done using a standard representation with boolean entries in an adjacency matrix.

1.2.2. *Shortest Path.* A more refined variant of this approach is to assign weights to the individual edges. For every pair of arrows, for which $\Diamond$ holds, we assign a weight of 1. For pairs of arrows, for which only $\circ$ holds, we assign a weight of 3. Pairs of arrows, for which $\circ$ does not hold, we assign a sufficiently high integer value, such as 6000, if it is unlikely to have 2000 traversals over arrows with a weight of 3. We can now apply a "shortest path" algorithm to a numerical adjacency matrix, using the lowest weight if multiple edges are present, allowing us to find paths between object classes, which require a minimum of modeling effort to be connected.

This "shortest path approach" can be used for a variety of analysis tasks, depending on the origin and semantics of the weights. Applications range from finding the memory or computationally most effective compositions, maximize the accuracy or other properties, if multiple paths lead to the same context type.

1.2.3. *Identifying Sub-Graphs and Bridges.* The transitive hull can help us identify unconnected sub-graphs, or bridges (which is computationally more expensive). This analysis can help us identify context types necessary for futher deductions.

1.2.4. *Node Degree.* The node degree for ingoing and outgoing edges can help us identify vital context types or object classes, which are very frequently used in the model. If nodes, which are not readily available, or must be computed with considerate computational effort, it may be desirable to redesign the entities involved.

Note that the applicability to different underlying categories allows greater flexibility. If we consider the node degree of a $\mathcal{CT}_T^{TOC}$ category, we can consider the availability of objects. If we apply a similar approach to a $\mathcal{CT}$ category, we can consider the computational cost of the method of a context type. This is a different approach than the shortest path, as the node degree allows us to study the dependency of the overall system on this context type.

1.3. Method and Type Weighting Scemes. As we have seen, we can make several interesting assumptions about the interpretation as a weighted graph. For this purpose, we will introduce weighting function templates for methods, type concepts and compound types.

Definition 1.3 (Weighting Functions). We now define a function template for weighting methods.

$$\omega_{Method} : \lceil Method \rfloor \to R, \tag{410}$$

where R is the set of real numbers.

Analogously, we will define a weighting function template for type concepts.

$$\omega_{TypeConcept} : \lceil TypeConcept \rfloor \to R. \tag{411}$$

We also introduce a weighting function template for compound types.

$$\omega_{CompoundType} : \lceil CompoundType \rfloor \to R. \tag{412}$$

Summarizing, we present a wrapper function ω, which collects the three weighting function templates presented above.

$$\omega : \lceil Method \rfloor \cup \lceil TypeConcept \rfloor \cup \lceil CompoundType \rfloor \to R \tag{413}$$
$$X \mapsto \begin{cases} \omega_{Method}(X) & \dots\ X \in \lceil Method \rfloor \\ \omega_{TypeConcept}(X) & \dots\ X \in \lceil TypeConcept \rfloor \\ \omega_{CompoundType}(X) & \dots\ X \in \lceil CompoundType \rfloor. \end{cases}$$

By building functions, which match these templates, we can map a typed object class view on context types to a weighted object class category, which we will introduce now.

Definition 1.4 (Weighted Object Class View on Context Types). For a given combined weighting function ω, we can define the categorical weighted object class

view on context types, denoted by $obj(\mathcal{CT}^{WOC})$, as follows.

$$obj(\mathcal{CT}^{WOC}) \subseteq \lceil TypedObjectClass \rfloor \times R \tag{414}$$

$$\begin{aligned} mor(\mathcal{CT}^{WOC}) &\subseteq \lceil TypedObjectClass \rfloor \times \\ &\times \lceil TypedObjectClass \rfloor \times \\ &\times R \times R \end{aligned} \tag{415}$$

$$\exists(A, B, \omega_m, \omega_t) \in mor(\mathcal{CT}^{WOC}) \Leftrightarrow A \xrightarrow[\omega_t]{\omega_m} B \tag{416}$$

$$\forall A, B, C \in obj(\mathcal{CT}^{WOC}) : A \xrightarrow[\omega_{t\circ}]{\omega_{m\circ}} C \ \textit{iff} \ A \xrightarrow[\omega_t]{\omega_m} B \ \wedge \ B \xrightarrow[\omega_{t'}]{\omega_{m'}} C \ , \tag{417}$$

with the following constraint for $m_\circ$ and $t_\circ$:

$$(m, A, B) \triangle t \wedge (m', B, C) \triangle t' \rightsquigarrow (m_\circ, A, C) \triangle t_\circ \quad \vee \tag{418}$$

$$\vee \qquad m_\circ = NullMethod \wedge t_\circ = \emptyset_{TC}, \tag{419}$$

and the method $NullMethod$ denotes that no composition is possible.

We can now define a translating functor from $obj(\mathcal{CT}^{WOC})$ to $obj(\mathcal{CT}^{TOC})$, denoted by Ω.

Definition 1.5 (A Functor from Object Class to Weighted Object Class View). For a given weighting function ω we define a functor Ω as follows.

$$\begin{aligned} \Omega : \quad \mathcal{CT}^{TOC} &\xrightarrow{\Omega} \mathcal{CT}^{WOC} \\ (A, T) &\longmapsto (A, T, \omega(T)) \\ A \xrightarrow[t]{m} B &\hookrightarrow \Omega(A) \xrightarrow[\omega(t)]{\omega(m)} \Omega(B) \ . \end{aligned} \tag{420}$$

We can now map an unweighted context type category with object class view to a fully weighted category, with weights for both nodes and edges for graph-based reasoning.

2. Creating Object Classes

In order to describe the workflow in a system, we need to be able to describe the very basic process of creating instances to object classes. We will describe different shades of this process, as we discriminate between initial instantiations and the objects, or instances, which can then be computed from the initial objects. We discriminate between these cases, as we always need initial data, or an initial source of data, from which further information can be computed.

2.1. Initial Instantiation. As most computations are based on some form of data, we need an initial amount of "raw material", from which we can then derive further insights. This initial data can be created by user interaction, read from a file or retrieved via TCP/IP. For our purposes, this is not of primary concern here, and will be discussed below. Again, we will use a notation of arrows to describe the system.

Initially, we will simply assume that an arrow,

$$A \longrightarrow B \ , \tag{421}$$

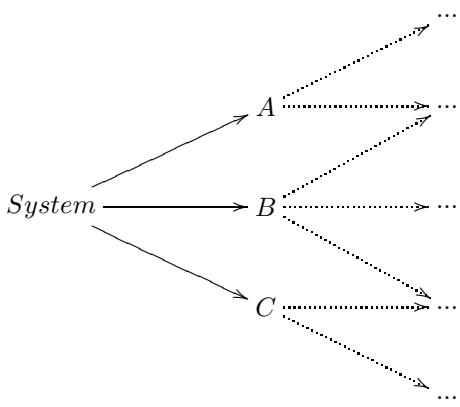

FIGURE 2. Initial Instantiation

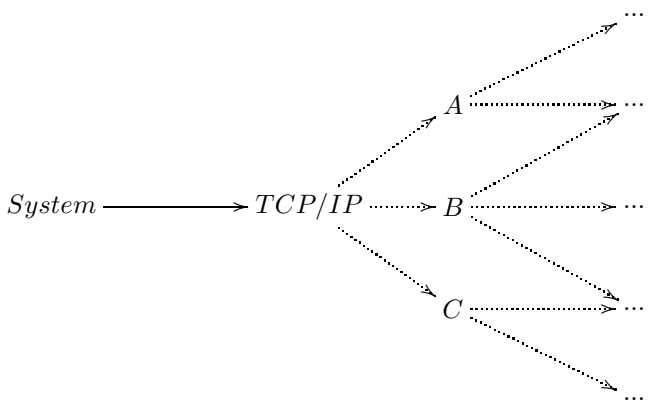

FIGURE 3. Initial Instantiation of a TCP/IP Resource

indicates that we can create object from the object class B from objects within object class A. This means that we describe the creation of instances on object class, or type level.

We will denote our concept of *initial instantiation* by using our *System* object class as a source of all concerned object classes. This basic modeling principle of is depicted in Figure 2. For our processing descriptions, it means that we can assume that every processing description graph is connected to the *System* object class.

2.2. Resource Object Classes. Our initial objects are often generated from files, databases or other sources of information. We will consider these *sources of information* to be resources. We now need a way of representing resources consistently in our formal model.

We will do so by introducing *resource object classes*. These object classes are tokens for specific resources. We can again use our notation of object class creation arrows to model this. Figure 3 depicts an initial instatiation, in which the system opens a TCP/IP connection and then creates further objects based on this connection.

2.3. Object Creation Categories. We can now use our arrow notation to model a category, which depicts the dependencies between object classes for the creation and computation of object classes. To do so, we require a function template for object creation, based on object class tuples.

DEFINITION 2.1 (Object Class Creation Function Template). We define a function template *create* for computing objects based on one or more object classes.

$$create : \lceil ObjectClass \rfloor^n \quad \rightarrow \quad \lceil Object \rfloor, \tag{422}$$

for all finite positive real-valued n,

$$n \in N \wedge n \geq 1 \wedge \exists m \in N : n < m. \tag{423}$$

Based on this function template, we can now derive a rule for checking the creation dependencies between object classes.

DEFINITION 2.2 (Object Class Creation Dependency Rule). For a given object creation function template *create*, we can define a rule $\oplus$ for representing creation dependencies amongst object classes as a binary relation.

$$\oplus \quad \subseteq \quad \lceil ObjectClass \rfloor \times \lceil ObjectClass \rfloor \tag{424}$$

$$\begin{aligned} A \oplus B \quad & \textit{holds, iff} \quad \exists C_1, ..., C_n \in \lceil ObjectClass \rfloor : \\ create(C_1, ..., C_n) \in B \quad & \wedge \quad \exists i \in N, 1 \leq i \leq n : A = C_i. \end{aligned} \tag{425}$$

We can now define a category, which depicts the creation dependencies in a system specification, based on a give *create* function template.

DEFINITION 2.3 (An Object Class Creation Dependency Category). For a given creation function *create*, we can define the category for depicting the creation dependencies amongst object classes, denoted by $\mathcal{CCD}$, as follows.

$$obj(\mathcal{CCD}) \quad \subseteq \quad \lceil ObjectClass \rfloor \tag{426}$$

$$mor(\mathcal{CCD}) \quad \subseteq \quad \lceil ObjectClass \rfloor \times \lceil ObjectClass \rfloor \tag{427}$$

$$A \rightarrow B \quad \Leftrightarrow \quad (A, B) \in mor(\mathcal{CCD}) \tag{428}$$

$$A \rightarrow B \in mor(\mathcal{CCD}) \quad \Leftrightarrow \quad A \oplus B \tag{429}$$

$$\begin{aligned} \forall A, B, C \in obj(\mathcal{CCD}) \quad & : \quad A \rightarrow C \ \textit{iff} \\ A \rightarrow B \quad & \wedge \quad B \rightarrow C. \end{aligned} \tag{430}$$

This new category allows us to study the dependencies between object classes, but for a real processing description, we will need to identify the function used to create objects of a given object class. As we may have multiple functions for creating members of an object class, we define a function template, which serves as a wrapper for a look-up of the label of the respective function, based on the available input object classes.

DEFINITION 2.4 (Object Class Creation Function Look-Up Template). We define a function template *lookup*, which returns the identifier of any function used to create a given object class, based on a number of other object classes.

$$lookup : \lceil ObjectClass \rfloor^n \times \lceil ObjectClass \rfloor \quad \rightarrow \quad \lceil Identifier \rfloor, \tag{431}$$

for all finite positive real-valued n,

$$n \in N \wedge n \geq 1 \wedge \exists m \in N : n < m. \tag{432}$$

This look-up function allows us to present a labeled variant of our creation dependency category, which serves as a basis for describing the object creation workflow of a system.

Definition 2.5 (An Object Class Creation Category). For a given creation function *create*, and a function label look-up function *lookup*, we can define the category for depicting the creation of object classes, denoted by $\mathcal{OCC}$, as follows.

$$obj(\mathcal{OCC}) \subseteq \lceil ObjectClass \rfloor \tag{433}$$

$$mor(\mathcal{OCC}) \subseteq \lceil ObjectClass \rfloor \times \lceil ObjectClass \rfloor \times \times \lceil Identifier \rfloor \tag{434}$$

$$A \xrightarrow{l} B \Leftrightarrow (A, B, l) \in mor(\mathcal{OCC}) \tag{435}$$

$$A \xrightarrow{l} B \in mor(\mathcal{OCC}) \Leftrightarrow A \oplus B \ \wedge \tag{436}$$

$$\forall\, C_1 \xrightarrow{l} B\,, \ ..., \ C_n \xrightarrow{l} B \in mor(\mathcal{OCC}) : l = lookup(C_1, ..., C_n, B) \tag{437}$$

$$\forall A, B, C \in obj(\mathcal{OCC}) : A \xrightarrow{composition} C \ \textit{iff} \ A \xrightarrow{l} B \wedge B \xrightarrow{l'} C\,. \tag{438}$$

This definition concludes our considerations about object class creation, allowing us to move on to a concise representation of methods.

3. Representing Methods

Up to this point, we have been working with a very open definition for methods.

$$\lceil Method \rfloor = \lceil X \rightarrow \lceil Context \rfloor \rfloor. \tag{439}$$

We will now craft a new definition, as a function template operating on object classes, to replace our initial definition.

Definition 3.1 (An Object Class Method Function Template). We now present an alternate definition of a method as a function template operating on several specific object classes.

$$\lceil Method \rfloor = \lceil ObjectClass^n \rightarrow \lceil Context \rfloor \rfloor, \tag{440}$$

for all finite positive real-valued n,

$$n \in N \wedge n \geq 1 \wedge \exists m \in N : n < m. \tag{441}$$

We can now use this function to create mappings between our object class creation category $\mathcal{OCC}$ and our initial category for context types. This function does not qualify as a functor, due to the different semantics of arrows in $\mathcal{OCC}$ and $\mathcal{CT}$.

Definition 3.2 (Representing Methods as Weak Functors). We now define a function $represent_{\mathcal{OCC},\mathcal{CT}}$ from multiple objects from $\mathcal{OCC}$ to $\mathcal{CT}$, which is based on method specifications according to the function template representation (see

Statement 440). This function can represent the dependencies of multiple methods between object classes and context types.

$$(442) represent_{\mathcal{OCC},\mathcal{AC}} : \lceil ObjectClass \rfloor^n \times \left\lceil \lceil Method \rfloor \right\rfloor \rightarrow \lceil ContextType \rfloor$$
$$(OCs, Methods) \mapsto \tau,$$

such that the following holds for τ:

$$(443) \qquad m = (OCs \rightarrow \lceil Context \rfloor) \in Methods \Rightarrow \tau = T(m(OCs)).$$

This representations of methods, namely as links between our object class creation categories and various context type based categories allow us to express rich system functionality, leading to processing and workflow specifications using the categories $\mathcal{OCC}$ and $\mathcal{CT}^{TOC}$, and a weak functor representations of methods, which can be crafted analogously as the representation on context types (see Definition 3.2), denoted by $represent_{\mathcal{OCC},\mathcal{CT}^{TOC}}$.

CHAPTER 8

Describing Systems

1. System Descriptions

Now that we have a vocabulary of describing context, data types and the workflow, we will examine how to describe systems in this language. We will do so by considering an initial minimum specification, which only reflects typed context types. We will then extend our description to include typed object classes and processing, and then present a final definition of a system description.

1.1. A Weak Initial Description. For an initial description of a system, we will only consider typed context types. If we recapture our definitions for context types and type concepts,

$$\lceil ContextType \rfloor = \lceil Method \rfloor \times \lceil ObjectClass \rfloor \times \lceil ObjectClass \rfloor, \tag{444}$$

$$\begin{aligned} \lceil TypeConcept \rfloor = \big\{ & T \,|\, T \in \lceil Value \rfloor \vee T \in \lceil Vector \rfloor \\ & \vee \; T \in \lceil Matrix \rfloor \vee T \in \lceil List \rfloor \\ & \vee \; T \in \lceil Assignment \rfloor \\ & \vee \; T \in \lceil AppliedValue \rfloor \vee T \in \lceil BoundValue \rfloor \\ & \vee \; T \in \lceil AppliedVector \rfloor \vee T \in \lceil BoundVector \rfloor \\ & \vee \; T \in \lceil AppliedMatrix \rfloor \vee T \in \lceil BoundMatrix \rfloor \\ & \vee \; T \in \lceil ObjectList \rfloor \vee T \in \lceil CTList \rfloor \big\}, \end{aligned} \tag{445}$$

and typed context types,

$$\lceil TypedContextType \rfloor = \lceil ContextType \rfloor \times \lceil TypeConcept \rfloor, \tag{446}$$

we can rewrite the expansion,

$$\begin{aligned} \lceil TypedContextType \rfloor = & \lceil Method \rfloor \times \lceil ObjectClass \rfloor^2 \\ & \times \lceil TypeConcept \rfloor. \end{aligned} \tag{447}$$

This complete denotation of our notion of typed context types helps us to identify the four basic sets, which are necessary for describing a system,

$$\lceil ObjectClass \rfloor \times \lceil Method \rfloor \times \lceil TypeConcept \rfloor \times \lceil TypedContextType \rfloor. \tag{448}$$

As object classes are defined as sets of objects,

$$\lceil ObjectClass \rfloor \subseteq \big\lceil \lceil Object \rfloor \big\rfloor, \tag{449}$$

thus describing objects, a description suffices for running systems, if we integrate the context entities,

$$\begin{aligned} \big\lceil \lceil ObjectClass \rfloor \big\rfloor \times \big\lceil \lceil Method \rfloor \big\rfloor \times \big\lceil \lceil TypeConcept \rfloor \big\rfloor \times \\ \times \big\lceil \lceil TypedContextType \rfloor \big\rfloor \times \big\lceil \lceil Context \rfloor \big\rfloor. \end{aligned} \tag{450}$$

Other relevant notions can be deduced from such a description,

$$\lceil TypedContextType\rfloor \rightsquigarrow \mu \tag{451}$$

$$\lceil TypedContextType\rfloor \rightsquigarrow \lceil ContextType\rfloor \tag{452}$$

$$\lceil ContextType\rfloor \rightsquigarrow \lceil ContextGroup\rfloor. \tag{453}$$

As our context group connectivity $\circ$, the T functor from context entities to context types and the Γ functor from types to groups are inherent to the model, we can also obtain our basic categories and the object class view on context types,

$$\lceil Context\rfloor \wedge \lceil ContextType\rfloor \wedge \lceil ContextGroup\rfloor \rightsquigarrow \mathcal{AC} \wedge \mathcal{CT} \wedge \mathcal{CG} \wedge \mathcal{CT}^{OC}. \tag{454}$$

Analogously, we can deduce the causality $\mathcal{TCT}_?$ and rule category $\mathcal{TCT}_!$ from our set of typed context types,

$$\lceil TypedContextTypes\rfloor \rightsquigarrow \mathcal{TCT}_? \wedge \mathcal{TCT}_!, \tag{455}$$

giving us enough premises to construct a context similarity category $\mathcal{ACS}$, if we define a suitable K, and any context traversal category $\mathcal{CTT}$.

The final addition to our initial system specification will now be a set of selection predicates p_S, allowing us to use all typed context type deduction possibilities. Summarizing, we can assume that the following deduction holds:

$$\begin{aligned}\lceil\lceil ObjectClass\rfloor\rfloor \times \lceil\lceil Method\rfloor\rfloor \times \lceil\lceil TypeConcept\rfloor\rfloor \times \\ \times\lceil\lceil TypedContextType\rfloor\rfloor \times \lceil\lceil Context\rfloor\rfloor \times \lceil\lceil p_S\rfloor\rfloor \\ \rightsquigarrow \\ \mathcal{AC} \wedge \mathcal{CT} \wedge \mathcal{CG} \wedge \mathcal{TCT}_? \wedge \mathcal{TCT}_!,\end{aligned} \tag{456}$$

and we have full typed context type inferencing capabilities.

1.2. Completing the Description. In order to include the remaining features of our context model, we now have to consider typed object classes, which can replace our set of object classes in the basic description, if we also include a set of compound types.

$$\begin{aligned}\lceil\lceil TypedObjectClass\rfloor\rfloor \times \lceil\lceil CompoundType\rfloor\rfloor \times \lceil\lceil Method\rfloor\rfloor \times \\ \times\lceil\lceil TypeConcept\rfloor\rfloor \times \lceil\lceil TypedContextType\rfloor\rfloor \times \lceil\lceil Context\rfloor\rfloor \times \lceil\lceil p_S\rfloor\rfloor.\end{aligned} \tag{457}$$

With this description, we can deduce our typed object class view on context types $\mathcal{CT}^{TOC}$ and any weighted object class view $\mathcal{CT}^{WOC}$, given a suitable weighting function ω.

Finally, if we use a set of *create* functions to describe the object class creation, we can also deduce our object class creation category $\mathcal{OCC}$ and object class creation dependency category $\mathcal{CCD}$.

$$\begin{aligned}\lceil\lceil TypedObjectClass\rfloor\rfloor \times \lceil\lceil CompoundType\rfloor\rfloor \times \lceil\lceil create\rfloor\rfloor \times \\ \times\lceil\lceil Method\rfloor\rfloor \times \lceil\lceil TypeConcept\rfloor\rfloor \times \\ \times\lceil\lceil TypedContextType\rfloor\rfloor \times \lceil\lceil Context\rfloor\rfloor \times \lceil\lceil p_S\rfloor\rfloor.\end{aligned} \tag{458}$$

If we now subsume our deductions, and consider the possibility to additionally construct a K monoid for similarity and ω weighting function, we can state the following:

$$\begin{aligned}(459)\qquad &\lceil\lceil TypedObjectClass\rfloor\rceil \times \lceil\lceil CompoundType\rfloor\rceil \times \lceil\lceil create\rfloor\rceil\times\\ &\times\lceil\lceil Method\rfloor\rceil \times \lceil\lceil TypeConcept\rfloor\rceil\times\\ &\times\lceil\lceil TypedContextType\rfloor\rceil \times \lceil\lceil Context\rfloor\rceil \times \lceil\lceil p_S\rfloor\rceil\\ &\wedge\\ (460)\qquad &K \wedge \omega\\ &\rightsquigarrow\\ (461)\qquad &\mathcal{AC} \wedge \mathcal{CT} \wedge \mathcal{CG} \wedge \mathcal{TCT}_? \wedge \mathcal{TCT}_! \wedge \mathcal{CT}^{TOC} \wedge \mathcal{OCC} \wedge \mathcal{CCD}\\ &\wedge\\ (462)\qquad &\mathcal{ACS} \wedge \mathcal{CT}^{WOC}.\end{aligned}$$

This means that we can construct all of our categorical notions from our description, which thus provides a solid basis for system descriptions.

Definition 1.1 (Atomic Component Descriptions). We can describe a system by stating all typed object classes and corresponding compound types, all methods, typed concepts, typed context types and context entities, and all selection predicates.

$$\begin{aligned}(463)\lceil AComponent\rfloor = \quad &\lceil\lceil TypedObjectClass\rfloor\rceil \times \lceil\lceil CompoundType\rfloor\rceil\times\\ &\times\lceil\lceil create\rfloor\rceil \times \lceil\lceil Method\rfloor\rceil \times \lceil\lceil TypeConcept\rfloor\rceil\times\\ &\times\lceil\lceil TypedContextType\rfloor\rceil \times \lceil\lceil Context\rfloor\rceil \times \lceil\lceil p_S\rfloor\rceil.\end{aligned}$$

Furthermore, we define the notion of a *finite* atomic component,

$$\begin{aligned}(464)\lceil AComponent\rceil = \quad &\lceil\lceil TypedObjectClass\rceil\rceil \times \lceil\lceil CompoundType\rceil\rceil\times\\ &\times\lceil\lceil create\rceil\rceil \times \lceil\lceil Method\rceil\rceil \times \lceil\lceil TypeConcept\rceil\rceil\times\\ &\times\lceil\lceil TypedContextType\rceil\rceil \times \lceil\lceil Context\rceil\rceil \times \lceil\lceil p_S\rceil\rceil.\end{aligned}$$

Using this atomic component description, we can describe all monolithic systems, which express all properties in one description. We will now briefly discuss the common issues of packaging and naming, to obtain the final definition of a system description.

2. Components and Packaging

As most systems are segmented into different components, which may well be designed by different authors or groups, no system description can avoid modeling means of packaging. We will do so by modelling a system as a *labeled root component*, which may have one or many *labeled sub-components*. To describe our labeled notions correctly, we need a few extra definitions on naming and labeling.

2.1. Naming and Labeling. If we want to name or label entities, we need a few initial definitions on strings. Traditionally, strings are defined as *elements of the powerset of an alphabet* $\sum$. We will use our stronger definition of *indexed sets* (see Section 2.3 Chapter 1) to describe strings of an alphabet $\sum$.

DEFINITION 2.1 (Identifiers). In this paper, we will consider an identifier to be defined as a string, based on an alphabet $\sum$.

$$\lceil Identifier \rfloor = \left\lceil \left\langle {}^{n} \sum \right\rangle \right\rfloor. \tag{465}$$

2.2. Final Component Definition and Systems. Usually, the root component will host any translative functionality between the intermediate sub-components and provide little functionality by itself. Contrary to most object-oriented approaches, the access to actual functionality does not have to be done over the root-component, but can also be done *sideways*, by directly accessing the sub-components.

In order to prevent any naming collisions, we will first introduce the concept of a labeled component.

DEFINITION 2.2 (Atomic Labeled Component Description). We define an atomic labeled component description as an atomic component description with an identifier,

$$\lceil IDAComponent \rfloor = \lceil Identifier \rfloor \times \lceil AComponent \rfloor \times \lceil N \rfloor, \tag{466}$$

and a finite variant,

$$\lceil IDAComponent \rceil = \lceil Identifier \rceil \times \lceil AComponent \rceil \times \lceil N \rfloor, \tag{467}$$

where $\lceil N \rfloor$ is a naming function template , which assigns an identifier to every element of the component,

$$\begin{aligned} \lceil N \rfloor = \Big\lceil \quad & \Big\lceil \lceil TypedObjectClass \rfloor \Big\rfloor \cup \Big\lceil \lceil CompoundType \rfloor \Big\rfloor \cup \\ & \cup \Big\lceil \lceil create \rfloor \Big\rfloor \cup \Big\lceil \lceil Method \rfloor \Big\rfloor \cup \Big\lceil \lceil TypeConcept \rfloor \Big\rfloor \cup \\ & \cup \Big\lceil \lceil TypedContextType \rfloor \Big\rfloor \cup \Big\lceil \lceil Context \rfloor \Big\rfloor \cup \Big\lceil \lceil p_S \rfloor \Big\rfloor \\ & \qquad \rightarrow \\ & \qquad \lceil Identifier \rfloor \qquad\qquad \Big\rfloor. \end{aligned} \tag{468}$$

Furthermore, we will define the following rule for referring to any entries from a labeled atomic component:

$$\begin{aligned} \forall (id, TypedOCs, Compounds, creates, methods, \\ TypeConcepts, TypedCTs, Contexts, \\ selectors, N) \in \lceil LAComponent \rfloor : \end{aligned} \tag{469}$$

$$id + \left\langle \{d\} \right\rangle + N(T) \quad \Leftrightarrow \quad N(T) \in TypedOCs \tag{470}$$

$$id + \left\langle \{d\} \right\rangle + N(C) \quad \Leftrightarrow \quad N(C) \in Compounds \tag{471}$$

$$id + \left\langle \{d\} \right\rangle + N(c) \quad \Leftrightarrow \quad N(c) \in creates \tag{472}$$

$$id + \left\langle \{d\} \right\rangle + N(m) \quad \Leftrightarrow \quad N(m) \in methods \tag{473}$$

$$id + \left\langle \{d\} \right\rangle + N(TC) \quad \Leftrightarrow \quad N(TC) \in TypeConcepts \tag{474}$$

$$id + \left\langle \{d\} \right\rangle + N(TCT) \quad \Leftrightarrow \quad N(TCT) \in TypedCTs \tag{475}$$

$$id + \left\langle \{d\} \right\rangle + N(AC) \quad \Leftrightarrow \quad N(AC) \in Contexts \tag{476}$$

$$id + \left\langle \{d\} \right\rangle + N(s) \quad \Leftrightarrow \quad N(s) \in selectors, \tag{477}$$

$$id + \left\langle \{n\} \right\rangle \quad \Leftrightarrow \quad N \in \lceil N \rfloor, \tag{478}$$

where $d \in \sum$ is a delimiter from the alphabet $\sum$ and $n \in \sum$ is a special character reserved for the naming function.

We will assume that this label translation is performed by the function *label*, allowing us to translate an atomic labeled component to a correctly labeled variant. This means that if we have a method m in a component A and a delimiter ".", we can assume the following:

$$label(m) \quad \rightarrow \quad "A.m" \tag{479}$$

$$"A.m" \quad \in \quad label(A). \tag{480}$$

We can now present the final definition of a component description, which now allows nested labeled components.

Definition 2.3 (Final Component Description). A component description consists of an atomic labeled component and a set of sub-component labels.

$$\lceil Component \rfloor = \lceil IDAComponent \rfloor \times \left\lceil \lceil Identifier \rfloor \right\rfloor. \tag{481}$$

We also define the notion of a finite component.

$$\lceil Component \rceil = \lceil IDAComponent \rceil \times \left\lceil \lceil Identifier \rceil \right\rceil. \tag{482}$$

If a component has no sub-components the set of labels is empty.

Summarizing, this leads to the definition of a system description.

Definition 2.4 (System Description). A context aware system, which is a system description using our context model, is a main component with a collection of all sub-components.

$$\lceil CAS \rfloor = \lceil Component \rfloor \times \left\lceil \lceil Component \rfloor \right\rfloor. \tag{483}$$

We have an analogous notion of a finite system,

$$\lceil CAS \rceil = \lceil Component \rceil \times \left\lceil \lceil Component \rceil \right\rceil. \tag{484}$$

The functionality we offer is the flattening of all sub-components with correct labels into a single component.

$$\begin{aligned} flatten : \lceil CAS \rfloor \quad &\rightarrow \quad \lceil Component \rfloor \\ (c_0, \{c_1, ..., c_n\}) \quad &\mapsto \quad join(\{c_0, label(c_1), ..., label(c_n)\}), \end{aligned} \tag{485}$$

where *join* combines all fields of a set of components,

$$\begin{aligned} join : \left\lceil \lceil Component \rfloor \right\rfloor \quad &\rightarrow \quad \lceil Component \rfloor \\ \{(c_{0,1}, ..., c_{0,9}), ..., (c_{n,1}, ..., c_{n,9})\} \quad &\mapsto \quad (c_{0,1} \cup ... \cup c_{n,1}, ..., c_{0,9} \cup ... \cup c_{n,9}). \end{aligned} \tag{486}$$

We can now use our considerations on components to describe more complex systems and increase the interoperability of descriptions. We have depicted an example in Figure 1.

2.3. Packaging. As the label of a component affects the names, which we use to refer to the elements, we can use component labels for packaging. These labels are not affected by the nesting of components, meaning that hierarchical package naming must be done by using the full hierarchical label for every component, regardless of any nested usage.

This means that a component must be explicitly labeled "A.B.C", even though it may be used as a sub-component of a component "A.B".

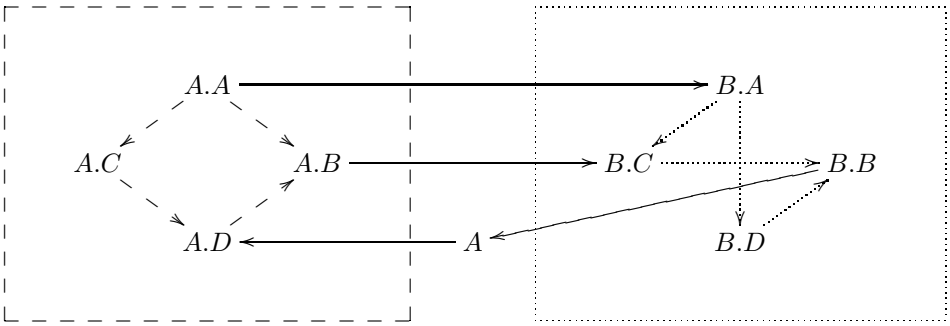

FIGURE 1. A Component with Two Sub-Components (Example)

CHAPTER 9

Basic Methodology

1. Introduction

In the field of system analysis, design and specification, more often than not, "[...] practice is leading theory [...]" (see [**2**]). For our formal model, this is only partly the case. The origins of the formal model presented in this report lie in the problems web-based systems face, when used in conjuction with, or based on modern methods from the field of artificial intelligence.

So clearly, this formal model has its roots in practice. But the presented formal model can lead to a new methodology in system analysis and development. It is our hope that the formal model presented here contains some power to feed back into practice. This chapter attempts to close the cycle from theory back to practice.

$$\textit{Practice} \rightsquigarrow \textit{Theory} \rightsquigarrow \textit{Practice} \rightsquigarrow \textbf{?} \tag{487}$$

2. Software System Functionality Deduction

We have studied the possibility to deduce new typed context types extensively in Chapter 4. Let us now study two examples of an actual deduction process, from the domain of natural language processing, and then roughly study the performance of this method.

2.1. A Vector Similarity Indexing System. We begin our modelling process by stating the object classes we wish to consider.

$Docs$ - A document is any form of textual document.
$Words$ - A word is an individual word or an n-gram (word n-tuple).
$Topics$ - A topic is a thematic concept.

Our first context type definition is that of a key-term frequency representation of a document. For our model, we can imagine this representation to be the *explicit representation of a functional dependency between documents and words.* The structure type is a vector with word index semantics. These considerations lead to the following context type:

$$(tf - construction, Docs, Words)\triangle(vec, Words). \tag{488}$$

We can also represent the concept of *artificial class vectors*, by using an object list of documents,

$$(class - assignment, Topics, Docs)\triangle(L, Docs), \tag{489}$$

then perform a context type substitution using our term frequency context type,

$$\begin{gathered}(class - vectors, Topics, Words)\\ \triangle\\ (class - assignment, Topics, Docs)\triangle(L, Docs),\end{gathered} \tag{490}$$

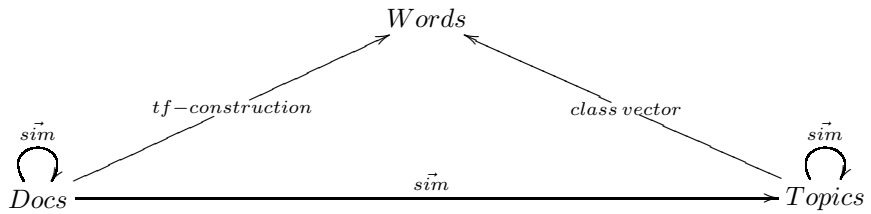

FIGURE 1. Document Class View of a Vector Similarity Indexing System

and compose the entire context type list to a matrix,

$$(doc/tf, Topics, Words)\triangle(M, Docs, Words), \tag{491}$$

which is very similar to the matrices used in latent semantic indexing and analysis, but also topic specific.

Given a suitable selection predicate p_S, we can select columns from this matrix to obtain the word frequencies for all class documents,

$$(col\ select, Topics, Words)\triangle(vec, Words), \tag{492}$$

and by selecting and merging components we obtain the individual words' average frequency in the class,

$$(ctlist\ subst, Topics, Words) \tag{493}$$

$$\triangle$$

$$(merge\ values, TopicsWords)\triangle(val, Words). \tag{494}$$

Finally, we can construct the artificial class vector using vector construction,

$$(tf\ class\ vector, Topics, Words)\triangle(vec, Words). \tag{495}$$

We can now deduce similarity between documents, between topics and between topics and documents:

$$doc\ sim = (vector\ similarity, Docs, Docs)\triangle(val, Words), \tag{496}$$

$$topic\ sim = (vector\ similarity, Topics, Topics)\triangle(val, Words), \tag{497}$$

$$doc/topic\ sim = (vector\ similarity, Docs, Topics)\triangle(val, Words). \tag{498}$$

We have now deduced a vector similarity indexing system (see Figure 1 for an object class view without deduction overhead).

2.2. An Extended Vector Similarity Indexing System. Now that we have studied the basic similarity measurements for vector similarity, we can turn to a higher level natural language processing system, which may be of interest for web systems. Again, we begin by stating our object classes:

$Docs$ - A document is any form of textual document.
$Words$ - A word is an individual word or an n-gram (word n-tuple).
$Topics$ - A topic is a thematic concept.
$Users$ - A user is a "customer" of our indexing system.

Initially, we assume that we have the basic similarity units from our initial vector similarity indexing system,

$$doc\,sim = (vector\,similarity, Docs, Docs)\triangle(val, Words), \tag{499}$$

$$topic\,sim = (vector\,similarity, Topics, Topics)\triangle(val, Words), \tag{500}$$

$$doc/topic\,sim = (vector\,similarity, Docs, Topics)\triangle(val, Words), \tag{501}$$

and the term frequency representations for documents and topics,

$$(tf - construction, Docs, Words)\triangle(vec, Words), \tag{502}$$

$$(tf\,class\,vector, Topics, Words)\triangle(vec, Words). \tag{503}$$

We now log all user interactions with documents, and create a numerical relevance metric for every document viewed by a user.

$$(user\,logging, Users, Docs)\triangle(val, Docs). \tag{504}$$

Using scalar multiplication, we can create a term-frequency vector, rescaled over user interest,

$$(scalar\,mult., Users, Docs)\triangle(vec, Docs). \tag{505}$$

As before with topics, we can now deduce a term frequency representation for users,

$$(vector\,construction, Users, Words)\triangle(vec, Words), \tag{506}$$

and derive further similarity metrics,

$$user/doc\,sim = (vector\,similarity, Users, Docs)\triangle(val, Words), \tag{507}$$

$$user/topic\,sim = (vector\,similarity, Users, Topics)\triangle(val, Words), \tag{508}$$

$$user\,sim = (vector\,similarity, Users, Users)\triangle(val, Words). \tag{509}$$

2.3. Deduction Analysis. As we have seen above, we have been able to deduce the key features of plain vector similarity, as well as a more advanced system functionality using very simple user profiles. To estimate the advantages and disadvantages of this process, let us consider our basic input and output during the process.

We have deduced several very interesting similarity metrics,

$$user\,sim = (vector\,similarity, Users, Users)\triangle(val, Words), \tag{510}$$

$$doc\,sim = (vector\,similarity, Docs, Docs)\triangle(val, Words), \tag{511}$$

$$topic\,sim = (vector\,similarity, Topics, Topics)\triangle(val, Words), \tag{512}$$

$$user/doc\,sim = (vector\,similarity, Users, Docs)\triangle(val, Words), \tag{513}$$

$$user/topic\,sim = (vector\,similarity, Users, Topics)\triangle(val, Words), \tag{514}$$

$$doc/topic\,sim = (vector\,similarity, Docs, Topics)\triangle(val, Words), \tag{515}$$

from very few initial context types,

$$(i) \quad (user\,logging, Users, Docs)\triangle(val, Docs) \tag{516}$$

$$(ii) \quad (tf - construction, Docs, Words)\triangle(vec, Words) \tag{517}$$

$$(iii) \quad (class - assignment, Topics, Docs)\triangle(L, Docs). \tag{518}$$

The additional "input" was the keep/discard decision process a fully expansive deduction process would require. As our set of rules (as in Chapter 4) in fact leads to a very high number of new context types, which need to be reviewed by the

user, an implementation of the system analysis approach, as we envision it, should be semi-automated, prompting the user for confirmation of new context types and various options regarding the *visibility* of the deductions in the tool's UI.

3. Concept Structure Descriptions

In the previous section, we have specified some basic context types to deduce further system functionality. In this section, we will not resort to any deduction processes, but rather use our context model to examine the structure of a system type on a theoretical level. As a well known example, we will study the structural requirements for a hypermedia system.

3.1. A Model for Hypermedia Systems. Once more, we begin our modeling phase with the specification of object classes.

$Users$ -	Users are site visitors.
$UserGroups$ -	A group of users is a set of individual users.
$Docs$ -	A document is any form of a digital media document, viewable with a web browser.
$Links$ -	Links are connections between pairs of documents.
$QueryLinks$ -	A query link is any link, which encloses a submission of parameters to a search.

We can now build a structural template for hypermedia systems. The context modeling approach will yield a high-level view on these systems. So instead of describing one particular hypermedia system, we will first use this high-level view to describe these systems as generic as possible.

3.1.1. *Linkage and Queries.* As a first step, we will define a context type, which describes the dependencies between documents and the links *from* this document.

$$(Document\,Linkage, Docs, Links)\triangle(OList, Links). \tag{519}$$

We will now describes the mapping from identified links to link targets.

$$(Default\,Link\,Target, Links, Docs)\triangle Links \rightarrow Docs. \tag{520}$$

This notion of having identified links allows us to treat links as "first-level objects".

In order to allow the modeling of user or knowledge adaptive hypermedia systems, we also allow alternate link targets.

$$(Alternate\,Link\,Targets, Links, Docs)\triangle(OList, Links). \tag{521}$$

We will also allow query links in documents, and build a suitable context type.

$$(Document\,Queries, Docs, QueryLinks)\triangle(OList, QueryLinks). \tag{522}$$

The actual functionality of a query will be hidden in an assignment.

$$(Query\,Processing, QueryLinks, Docs)\triangle QueryLinks \rightarrow Docs. \tag{523}$$

These are our basic context types for describing linkage and queries, but we also need some basic transformations to describe a contemporary web applications.

3.1.2. *Users.* Users can very easily be assigned to user groups, even though the application specific decision on *how* to group users can be very complex. The context type for user grouping is very independent from this.

(524) $(User\,Grouping, Users, UserGroups)\triangle Users \rightarrow UserGroups$

We also need to formalize the current document of a user.

(525) $(User\,Location, Users, Docs)\triangle Users \rightarrow Docs.$

Regarding user profiling, we initially consider the naviation sequence over documents.

(526) $(User\,Document\,Trail, Users, Docs)\triangle(OList, Docs).$

We then consider the click sequence over links,

(527) $(User\,Link\,Sequence, Users, Links)\triangle(OList, Links),$

which also respects any alternate link targets.

Finally, we apply the same concept to queries.

(528) $(User\,Query\,Sequence, Users, QueryLinks)\triangle(OList, QueryLinks).$

By applying our context type definitions so far, we can now model navigation.

3.1.3. *Navigation.* As a user enters the site, we can assign an internal document as an initial value of our user location context (see Statement 525).

If our system is relying on single link targets only, we can use the assignment in the link target context type (see Statement 520) to update the user's location after using the link, giving us a very simple concept of navigation to begin with.

But for selecting a link from the list of links, which is associated with every document, we need a boolean predicate for selection. Using a predicate, which yields true for a link, which has been clicked on, we can select the appropriate link from the typed list in the context type in Statement 519.

If we are dealing with a more adaptive hypermedia system, we want to exploit our alternate link targets. To do so, we need another predicate for selecting the appropriate alternate link target. This can be a selection in a pop-up window or any form of artificial intelligence support. This approach leads to a much greated flexibility (see Figure 2).

3.2. Description Analysis. Our description of a hypermedia system can represent most forms of navigation. It still leaves several issues open, but it can serve as a *template* for a hypermedia system. We can thus assume that the performance of our description depends on its performance as a template.

If we view this template isolated from other system descriptions, the template may provide a basis for further structural examination, but for greater benefit, we have to view it together with other system descriptions.

4. Connecting Systems

We now want to study and exploit the possibilities of describing *more than one* system in our formal language. For this purpose, we will discern two cases:

- Connecting two or more systems.
- Replacing a context type in a system, based on the context types of other systems (arrow substitution).

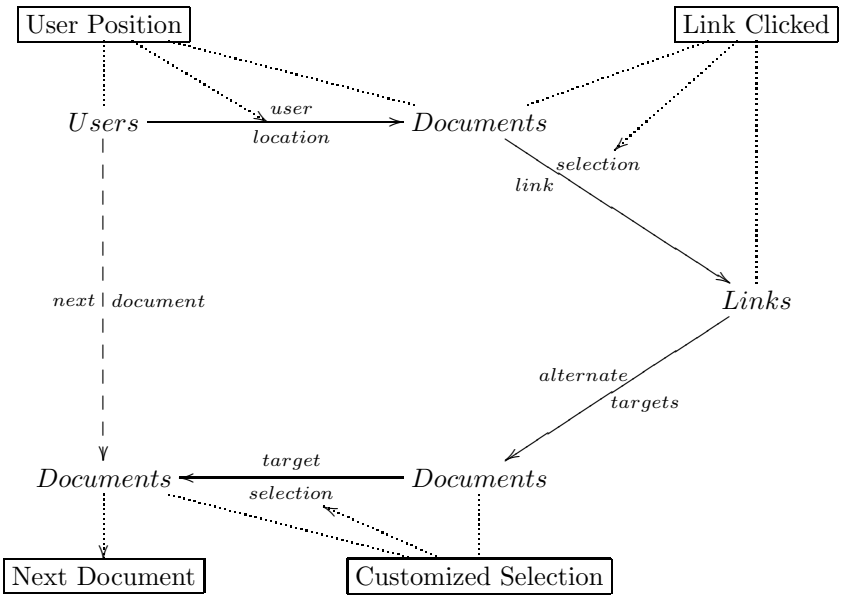

FIGURE 2. Adaptive Navigation Sketch

4.1. Connecting Systems. The task of connecting systems is identical to our consideration about nested sub-components, as discussed in Section 2, Chapter 8. In fact, we have already connected two systems in Section 2.2 in this chapter.

In the extended vector similarity example, we have simply created a one-on-one context type between the *Docs*, *Words* and *Topics* object classes, which corresponds to an identity morphism. These three additional and trivial context types have been sufficient to connect the basic vector similarity indexing system and the basic mechanisms for user related retrieval techniques.

On a more general level, we can consider all systems as sub-components the new combined system, and connect them by adding context types to the base component.

4.2. Arrow Substitution. Once we have connected several systems, we can use our basic deductions and infer new context types. During this inferencing process, it may well be the case the we obtain multiple arrows between pairs of object classes. If a developer chooses to do so, old arrows might be dropped and are replaced by the new context types, because these constitute a similar functionality, and are more suitable for the respective task.

If one of the sub-systems is a concept structure description, and is only a template for a special kind of system (web system, spreadsheet system, etc.), we can show that our *other* combination of systems is capable of fulfilling one of the tasks necessary for the system described by the *template*, if we can replace a context type in the template with a deduced context type, which is based on the other systems.

This means that arrow substitution can be used to check if a combination of systems can solve a specific task in a more general application domain, if we have a concept structure description for the application domain.

4.2.1. *Arrow Substitution in the Hypermedia Model (Example).* We will now study an example of the arrow substitution process. If we use our hypermedia model in conjunction with our vector similarity indexing system, we can derive a term-frequency representation for every link from our default link target,

$$(object\,translation, Links, Words)\triangle(vec, Words). \tag{529}$$

If we now use vector similarity between links and documents,

$$(vector\,similarity, Links, Docs)\triangle(val, Words), \tag{530}$$

we can obtain document recommendations via context type list construction and list selection,

$$(list\,selection, Links, Docs)\triangle(OList, Docs). \tag{531}$$

We can now use this context type to replace our generic token for alternate link targets,

$$(Alternate\,Link\,Targets, Links, Docs)\triangle(OList, Links) \tag{532}$$
$$\leftarrow$$
$$(list\,selection, Links, Docs)\triangle(OList, Docs).$$

We have shown that, given main link targets, we can use vector similarity to determine alternate link targets.

Bibliography

[1] Dublin core metadata element set, version 1.1: Reference description, April 2003.

[2] R. Glass. Editor's corner: Theory + practice: A disturbing example. *J. Systems Software*, 25:125–126, 1994.

[3] J. McCarthy and S. Buvac. Formalizing context (expanded notes). In *Working Papers of the AAAI Fall Symposium on Context in Knowledge Representation and Natural Language*, pages 99–135. American Association for Artificial Intelligence, 1997.

[4] Marvin Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211–280. McGraw-Hill, New York, 1975.

[5] J. van Oosten. *Basic Category Theory*. BRICS Lecture Series LS-95-1, University of Aarhus, Denmark, January 1995.